Implementación eficiente de construcciones de alto nivel para la programación concurrente

por Damián Barsotti

Presentado ante la Facultad de Matemática, Astronomía y Física como parte de los requerimientos para la obtención del grado de Doctor en Ciencias de la Computación de la

Universidad Nacional de Córdoba

Julio, 2011

@FaMAF – UNC 2011

Director: Javier Oscar Blanco

Resumen

En el campo científico y tecnológico actual es de amplia aceptación que el desarrollo de programas concurrentes es una tarea difícil y además necesaria. Aunque el interés en esta actividad no es nuevo, ha tomado particular relevancia en los últimos tiempos debido al desarrollo y popularidad de las arquitecturas paralelas.

Con el fin de aprovechar estas arquitecturas se han propuesto varias construcciones de alto nivel para los lenguajes de programación, que funcionan como abstracciones sobre el hardware o el sistema operativo subyacente. En este sentido, las construcciones clásicas denominadas regiones críticas condicionales y su posterior evolución en monitores de señalamiento automático, proveen abstracciones sobre el hardware o el sistema operativo subyacente que posibilitan el desarrollo de programas con variables compartidas y mecanismos de sincronización entre distintos procesos. Las mismas son herramientas interesante para desarrollar programas concurrentes de manera simple y de forma correcta. Lamentablemente estas propuestas no han sido incluidas como parte del diseño de los lenguajes de programación debido principalmente a problemas de eficiencia en sus implementaciones.

A partir de las nuevas posibilidades abiertas por el desarrollo reciente de las herramientas denominadas en forma genérica demostradores automáticos de teoremas, aplicadas dentro de técnicas de interpretación abstracta, se presenta una nueva perspectiva en la implementación eficiente de construcciones de alto nivel para la programación concurrente. El trabajo de tesis doctoral aquí presentado aborda esta problemática mediante la adaptación de estas metodologías, no ya para verificar programas, si no para modificarlos haciéndolos más eficientes. Más específicamente, desarrollaremos métodos automáticos para mejorar las implementaciones de regiones críticas condicionales y monitores con señalamiento automático mediante el uso de aquellas técnicas y herramientas. De tal manera, intentaremos probar la factibilidad de estas construcciones clásicas para la programación concurrente.

Palabras claves: programación concurrente, probador de teoremas, SMT solver, optimización, región crítica condicional, monitor, señal, semáforo binario dividido.

PACS: D.1.3,D.2.4,D.3.3,D.4.8,I.2.2,I.2.3

Abstract

In computer science and related technologies it is widely acknowledged that concurrent programming is a difficult but necessary task. Although the interest in this activity is not new, it has taken a particular relevance in recent times due to the development and popularity of parallel architectures.

In order to take advantage of these architectures, several high-level programming languages constructions have been proposed, which serve as abstractions of the hardware or the underlying operating system. In this sense, the classical structures called Conditional Critical Regions and their subsequent evolution in Automatic-signal Monitors, provide abstractions which enable the development of programs with shared variables and synchronization mechanisms between processes. Furthermore, they are interesting tools to develop concurrent programs in a simple and correct way. Unfortunately, these proposals have not been included as part of the design of programming languages mostly due to efficiency problems in their implementations.

The new possibilities that emerge from recent developments of certain tools known as automatic theorem provers, mainly when applied together with abstract interpretation techniques, open new perspectives on the efficient implementation of high-level constructs for concurrent programming. This PhD thesis addresses this problem by adapting some methods usually used to verify programs in order to transform them into more efficient ones. In summary, we will develop automatic methods to mechanically improve implementations of Conditional Critical Regions and Automatic-signal monitors through a novel use of these techniques and tools. As a consequence, we will re-asses the feasibility of these classical constructions for concurrent programming.

Keywords: concurrent programming, theorem prover, SMT solver, optimization, conditional critical region, monitor, signal, split binary semaphore, compiler.

A la Kuka y Lucio.

Agradecimientos

Quisiera agradecer principalmente a Javier Blanco, no solo por su labor como director de este trabajo, sino también por la amistad que compartimos a lo largo de estos años.

El camino que me condujo a finalizar esta tesis doctoral comenzó durante mi carrera de grado. Por la ayuda que me brindaron para poder finalizarla, quisiera agradecer a Rosita Wachenchauzer y a Sergio Cannas.

Por el apoyo que me brindó durante el periodo de desarrollo de esta tesis, quisiera agradecer especialmente a Leonor Prensa Nieto. Ella me ofreció la posibilidad de trabajar juntos brindándome su experiencia y conocimientos sobre probadores de teoremas lo cual abrió la puerta para el desarrollo de este trabajo.

Mi más grande agradecimiento es a la Kuka. Ella es mi compañera elegida en este camino que hicimos juntos. También quisiera agradecer a mi hermano Mariano y a mis padres, Romano y Jovita.

Índice general

Agradecimientos **VII**

Índice general **IX**

1 Sistemas de Transiciones **1**
1.1 Formalización 1
1.1.1 Notación 2
1.1.2 Expresiones cuantificadas 2
1.2 Lógica de predicados 4
1.2.1 Sintaxis 7
1.3 Sentencias de programa 7
1.3.1 Sintaxis 10
1.4 Transformadores de predicados 10
1.4.1 Strongest postcondition 11
1.4.2 Weakest liberal precondition 12
1.4.3 Propiedades de los transformadores 13
1.5 Sistemas de transiciones 17
1.5.1 Gráfico de transiciones 20
1.5.2 Sistemas de transiciones ejecutables 21
1.6 Transformadores de predicados en ST 22
1.6.1 Strongest postcondition 23
1.6.2 Weakest liberal precondition 24
1.6.3 Propiedades de los transformadores 26
1.7 Semántica operacional 28
Apéndice 1.A Propiedades de expresiones cuantificadas 34

2 Invariantes **39**
2.1 Puntos Fijos 39
2.1.1 Estructuras algebraicas y juntividad 39
2.1.2 Puntos fijos 42
2.2 Invariantes 43
2.3 Invariantes y puntos fijos 47
2.3.1 Propagación hacia adelante 48
2.3.2 Propagación hacia atrás 53
2.3.3 Propiedades 58
2.3.4 Método de demostración de invariancia 60

3 Generación de invariantes lineales 67
3.1 Interpretación Abstracta . . . 67
3.1.1 Dominio semántico concreto . . . 67
3.1.2 Dominio semántico abstracto y abstracción . . . 68
3.1.3 Función semántica abstracta . . . 69
3.1.4 Widening . . . 71
3.2 Poliedros convexos cerrados . . . 74
3.2.1 Representación algebraica . . . 75
3.2.2 Representación geométrica . . . 76
3.2.3 Conversión de marco a sistema de restricciones . . . 80
3.2.4 Conversión de sistema de restricciones a marco . . . 84
3.3 Invariantes en el dominio de poliedros convexos . . . 89
3.3.1 Abstracción . . . 89
3.3.2 Función semántica abstracta . . . 92
3.3.3 Sentencias abstractas . . . 93
3.3.4 Transformador de poliedros . . . 97
3.3.5 Sistemas de transiciones abstractos . . . 99
3.3.6 Widening . . . 100
3.3.7 Invariantes lineales . . . 101
3.3.8 Implementación . . . 102

4 Implementación de regiones críticas condicionales 103
4.1 Regiones críticas condicionales . . . 104
4.2 Técnica SBD . . . 106
4.3 Antecedentes . . . 117
4.4 SBD como sistema de transiciones . . . 118
4.5 Eliminación de guardas . . . 128
4.6 Mejoras del método . . . 132
4.6.1 Simplificación con invariante SBD . . . 132
4.6.2 Simplificación del sistema de transiciones . . . 133
4.6.3 Simplificación del invariante candidato . . . 136
4.6.4 Simplificación de fórmulas . . . 137
4.7 Implementación . . . 138
4.7.1 Generador de Transiciones . . . 138
4.7.2 Verificador de Invariantes . . . 140
4.8 Resultados . . . 140
4.8.1 Productor/Consumidor en buffer acotado . . . 140
4.8.2 Contraejemplo Schmid . . . 143
4.8.3 Semáforos generales . . . 144
4.8.4 Lectores y escritores . . . 144
4.8.5 Productor/Consumidor goloso . . . 145
4.8.6 Productor/Consumidor goloso M . . . 147
4.9 Conclusiones y trabajos futuros . . . 148

5 Implementación de monitores con señalamiento automático 151
5.1 Monitores . . . 152
5.1.1 Señalamiento explícito . . . 153
5.1.2 Señalamiento automático . . . 157
5.2 Antecedentes . . . 161
5.3 Implementación de monitores con señalamiento automático . . . 161

5.4 Monitores como sistema de transiciones 162
5.5 Eliminación de señalizaciones 176
5.5.1 Invariante candidato 176
5.5.2 Invariante inicial 182
5.6 Mejoras del método 183
5.7 Implementación 184
5.7.1 Etapa de inicialización 185
5.7.2 Etapa de eliminación de señalizaciones 186
5.8 Resultados 187
5.8.1 Semáforos generales 187
5.8.2 Productor/Consumidor en buffer acotado 188
5.8.3 Sincronización de Fase 190
5.8.4 Lectores y escritores 191
5.8.5 Baño unisex 193
5.8.6 Barrera 194
5.8.7 Productor/Consumidor goloso M 196
5.8.8 Rendezvous 197
5.8.9 Resumen 199
5.9 Conclusiones y trabajos futuros 200
Apéndice 5.A Reglas de derivación de transiciones 203
Apéndice 5.B Transiciones Productor/Consumidor 204
Apéndice 5.C Demostración Teorema 5.4 208

6 Conclusiones y trabajos futuros 217

Índice alfabético 219

Bibliografía 223

Capítulo 1

Sistemas de Transiciones

En este capítulo formalizaremos el concepto de *sistemas de transiciones* y mostraremos cómo puede utilizarse para describir programas no estructurados. Antes de ello presentaremos la lógica que nos permitirá esta formalización. La misma estará basada principalmente en [BAW98] y las demostraciones seguirán el estilo de [DS90, BSB08]. A continuación explicaremos la notación y el estilo de formalización que utilizaremos.

1.1 Formalización

Existen principalmente dos enfoques para formalizar la semántica de programas. El enfoque denominado "*deep embedding*" consiste en representar la sintaxis abstracta de los programas, generalmente mediante gramáticas o tipos de datos inductivos, y a posteriori asignarle una semántica. El mismo resulta útil cuando se quiera razonar sobre el lenguaje en si mismo a un nivel meta-teórico, generalmente haciendo inducción sobre la sintaxis.

Otro enfoque es el denominado "*shallow embedding*" en el cual los objetos sintácticos (programas y predicados sobre los mismos) son representados directamente de manera semántica mediante lógicas de alto orden, proporcionando además, mecanismos de traducción entre ambos contextos. De esta forma, el razonamiento sobre programas no estructurados (programas concurrentes, con sentencias **goto**, etc.) se simplifica ya que se trabaja directamente sobre la semántica[1].

En nuestro trabajo utilizaremos el segundo enfoque. Esta decisión se debe a que la formalización de sistemas de transiciones es difícil de realizar con las estructuras sintácticas de carácter recursivo propias del primer enfoque. Además, como veremos en el próximo capítulo, esta decisión simplifica la aplicación de la teoría de puntos fijos (de carácter claramente semántico) a los transformadores de predicados sobre sistemas de transiciones.

En este contexto por ejemplo un predicado será directamente denotado por un conjunto de estados y una sentencia de un programa será una relación entre los mismos. De esta manera, todas las entidades (predicados, programas, etc.) serán denotados a un nivel semántico. Cuando necesitemos hacer mención a la

[1] Un ejemplo de este tipo de formalización puede encontrarse en [NPN99].

sintaxis de programas o predicados, esta funcionará como una abreviación[2] de su semántica (más adelante profundizaremos sobre estas nociones). En estos casos daremos la traducción explícita entre los dos contextos, como lo permite el enfoque utilizado.

1.1.1 Notación

En nuestro trabajo seguiremos principalmente la notación de la lógica de alto orden HOL[3] desarrollada en [BAW98]. De este marco conceptual utilizaremos la notación para términos booleanos, donde el tipo Bool denota el reticulado de valores de verdad $\{\mathsf{T},\mathsf{F}\}$, junto con sus operaciones usuales $\wedge$, $\vee$, $\equiv$, $\Rightarrow$, $\neg$. La aplicación de función será representada por un punto, por ejemplo $f.x$. El dominio de una función f será denotado por $\mathsf{Dom}.f$.

Los conjuntos serán denotados por funciones características como usualmente se expresan en la lógica de alto orden: un conjunto con elementos en Γ será una función de tipo $\Gamma \to \mathsf{Bool}$. También utilizaremos la notación usual de teoría de conjuntos $\{\sigma : \Gamma \mid E\}$ en vez de $\langle \lambda\sigma : \Gamma \bullet E\rangle$ con E un término booleano. La cardinalidad de un conjunto C será denotada por $\mathsf{Card}.C$ y el conjunto de sus partes como $\wp.C$. El *operador de elección* será denotado por ϵ (especificado en [BAW98, pág. 135]).

Las relaciones n-arias serán denotadas de la forma usual en HOL como funciones que toman n elementos y devuelven un elemento en Bool. Por ejemplo, si una relación r relaciona elementos de los dominios $\Gamma_1, \cdots, \Gamma_n$ su tipo será $r : \Gamma_1 \to \cdots \to \Gamma_n \to \mathsf{Bool}$. Para una relación binaria su dominio será denotado $\mathsf{Dom}.r$.

Cuando introduzcamos definiciones utilizaremos el símbolo $\doteq$, por ejemplo $\mathsf{Bool} \doteq \{\mathsf{T},\mathsf{F}\}$. Utilizaremos además *definiciones locales*[4] encerradas entre los símbolos $[\![\ ]\!]$. Las mismas servirán para definir objetos solo dentro del ámbito de la expresión que la precede.

En el trabajo anteriormente referenciado se incluyen varias notaciones para diferentes tipos de cuantificación. En esta tesis unificaremos estas expresiones como se describe a continuación.

1.1.2 Expresiones cuantificadas

En diversos contextos (aritmética, lógica, teoría de conjuntos, lenguajes de programación, etc.) aparece cierta noción de cuantificación, entendiéndose por esto al uso de variables formales con un alcance delimitado explícitamente, las cuales pueden usarse para construir expresiones dependientes de ellas pero solo dentro de ese alcance. Por ejemplo, en el ámbito de la aritmética suele denotarse la suma de los primeros n numero impares como

$$\sum_{i=0}^{n-1} 2 * i + 1 \ \ .$$

En este trabajo se utilizará una notación unificada para este tipo de expresiones. La notación debe tener en cuenta al operador con el cual se cuantifica

[2] También llamada "syntactic sugar" de la semántica.
[3] Siglas de High Orden Logic.
[4] Similar a las clausulas **where** del lenguaje Haskell.

(en nuestro ejemplo la suma), las variables que van a usarse para crear las expresiones (i en el ejemplo), el rango de variación de estas variables ($0 \leqslant i < n$) y la expresión dependiente de las variables que define los términos de la cuantificación (en nuestro ejemplo $2 * i + 1$).

Una *expresión cuantificadas* será entonces un término de la siguiente forma:

$$\langle \oplus i \ : R : \ T \rangle$$

donde $_ \oplus _ : \Gamma \to \Gamma \to \Gamma$ designa un operador asociativo y conmutativo (por ejemplo, $+$, $\vee$, $\wedge$, Max, Min, etc.) de tipo Γ, i denota una secuencia de variables lógicas, $R : \mathsf{Bool}$ es un término booleano denominado *rango de especificación* y $T : \Gamma$ es un término de tipo acorde al operador $\oplus$ llamado *término de cuantificación*. La ocurrencia de i junto al operador suele llamarse *variable de cuantificación* o *dummy*. Cuando necesitemos hacer referencia explícita a la variable cuantificada en el rango o en el término escribiremos $R.i$ y $T.i$ respectivamente. El tipo de la variable puede en general inferirse del contexto, en caso contrario se lo definirá explícitamente.

De esta forma, la sumatoria en el ejemplo anterior puede escribirse como:

$$\langle + i \ : 0 \leqslant i < n : \ 2 * i + 1 \rangle$$

ya que el operador asociativo y conmutativo es la suma aritmética, o también utilizando el símbolo de sumatoria usual:

$$\langle \textstyle\sum i \ : 0 \leqslant i < n : \ 2 * i + 1 \rangle \ .$$

Para el caso de la conjunción $\wedge$ y la disyunción $\vee$ utilizaremos los símbolos habituales $\forall$ y $\exists$, esto es:

$$\langle \forall i \ : R : \ T \rangle \text{ y } \langle \exists i \ : R : \ T \rangle$$

respectivamente[5].

Esta notación tiene varias ventajas sobre la utilizada usualmente en matemática. Por un lado los símbolos $\langle \ \ \rangle$ determinan exactamente el alcance de la variable. Por otro lado, en matemática es bastante engorroso escribir rangos que no sean intervalos de número naturales. Las expresiones cuantificadas presentadas aquí admiten cualquier expresión booleana como rango, permitiendo además usar más de una variable cuantificada de manera natural, lo cual es casi imposible de escribir con la notación tradicional. Aún para sumatorias con una sola variable surgen algunos problemas. Tomemos por ejemplo la suma de los primeros n cuadrados de números distintos de m. Con la notación usual el rango de la cuantificación suele escribirse apilando las condiciones debajo del símbolo de sumatoria:

$$\sum_{\substack{i=0 \\ i \neq m}}^{n-1} i^2$$

lo cual resulta engorroso. Además puede llegar a ser ambiguo si se desea utilizar otros operadores lógicos o términos booleanos más complejos en el rango de

[5]La notación para la cuantificación universal y existencial es equivalente a la utilizada en [BAW98, pág. 131] denominada *bounded quantification*. Las mismas tienen la forma $(\forall i \mid R \cdot T)$ y $(\exists i \mid R \cdot T)$ respectivamente.

la sumatoria. En nuestra notación la sumatoria puede escribirse en forma más clara como:

$$\left\langle \sum i \ : 0 \leqslant i < n \wedge i \neq m : \ i^2 \right\rangle$$

permitiendo además rangos complejos en la cuantificación como por ejemplo

$$\left\langle \sum i \ : 0 \leqslant i < n \wedge (i \neq m \ \vee \ m = 0) : \ i^2 \right\rangle \ .$$

En [Dij00, Dij80a] se analizan las ventajas de esta notación y otras seguidas en esta tesis.

En algunas ocasiones no se desea restringir el rango de especificación al conjunto de variables en un término booleano R, como hemos escrito más arriba. En estos casos escribiremos $\langle \oplus i \ :: \ T \rangle$, entendiéndose que el rango abarca a todos los elementos del tipo de i.

Existe una serie de propiedades que sirven para manipular expresiones cuantificadas, las mismas pueden encontrarse en el apéndice 1.A (pág. 34) y en [BSB08, capítulos 5 y 6], las cuales serán utilizadas en las demostraciones.

En nuestro trabajo utilizaremos frecuentemente esta notación para denotar ínfimos y supremos sobre reticulados completos. Sea el reticulado $(\Gamma, \sqcup, \sqcap)$ entonces denotaremos

$$\langle \bigsqcup i \ : R : \ T \rangle \text{ y } \langle \bigsqcap i \ : R : \ T \rangle$$

con $R : \mathsf{Bool}$ y $T : \Gamma$, al supremo y al ínfimo respectivamente de los elementos denotados por T restringidos al rango R. También utilizaremos esta notación para posets en general siempre que los ínfimos y supremos estén bien definidos por el rango.

1.2 Lógica de predicados

Los programas imperativos trabajan realizando cambios sucesivos en la memoria. Estas diferentes configuraciones de memoria pueden ser representadas de forma abstracta como un conjunto Σ al cual denominaremos *espacio de estados* y a cada elemento del mismo como *estado*.

Nota: Como vemos, los estados son denotados de forma abstracta como un conjunto, sin hacer hincapié en las particularidades de sus elementos. Esto nos permitirá más adelante utilizar los resultados expuestos sobre distintos modelos. Un modelo clásico de estados son las funciones de un conjunto de variables a un conjunto de valores, por ejemplo:

$$\Sigma \ \doteq \ \mathsf{Var} \to \mathbb{Z} \ .$$

En este modelo las zonas de memoria son símbolos de variables y los valores serán elementos de $\mathbb{Z}$.

Otro modelo de estados son las n-uplas pertenecientes a $\mathbb{N}^n$, $\mathbb{Z}^n$ o $\mathbb{Q}^n$, por ejemplo:

$$\Sigma \ \doteq \ \mathbb{Z}^n \ .$$

En este modelo se presupone que cada coordenada de una n-upla representa el valor de una variable determinada. Esta representación de la memoria será útil

cuando la cantidad de variables esté acotada por n. Cuando las variables del programa tengan distintos tipos esta representación del espacio de estados puede ser extendida fácilmente utilizando productos de distintos conjuntos (por ej. $\mathbb{N} \times \mathbb{Q} \times \mathbb{Q}$).

En este capítulo, cuando necesitemos hacer mención a la sintaxis de predicados y programas (ver *shallow embedding* en la sección anterior) utilizaremos el primer modelo. Esto nos permitirá, además de simplificar la exposición, expresar los nombres de variables de programas de forma explícita. $\square$

Sea Bool el reticulado de valores de verdad ($\mathsf{Bool} \doteq \{\mathsf{T}, \mathsf{F}\}$) junto con sus operaciones usuales ($\wedge$, $\vee$, $\equiv$, $\Rightarrow$, $\neg$). Definiremos como *predicado* a una función con tipo $\Sigma \to \mathsf{Bool}$. El conjunto de todas estas funciones lo denominaremos

$$\mathsf{Pred} \doteq \Sigma \to \mathsf{Bool} \quad .$$

De esta forma los predicados son la extensión puntual del reticulado Bool. A partir de esta caracterización se puede deducir que es un reticulado booleano completo ya que hereda las propiedades de Bool [BAW98, lema 2.5] y además es atómico[6] [BAW98, corolario 7.1].

Dados $p, q, r \in \mathsf{Pred}$ y $\sigma \in \Sigma$ definimos las siguientes constantes y operaciones en Pred como extensiones puntuales de las operaciones en Bool:

$$\begin{array}{rcll}
\mathsf{true}.\sigma & \doteq & \mathsf{T} & (\top \text{ en } \mathsf{Pred}) \\
\mathsf{false}.\sigma & \doteq & \mathsf{F} & (\bot \text{ en } \mathsf{Pred}) \\
(p \cap q).\sigma & \doteq & p.\sigma \wedge q.\sigma & (\sqcap \text{ en } \mathsf{Pred}) \\
(p \cup q).\sigma & \doteq & p.\sigma \vee q.\sigma & (\sqcup \text{ en } \mathsf{Pred}) \\
(\neg p).\sigma & \doteq & \neg p.\sigma & (\neg \text{ en } \mathsf{Pred}) \\
(p \equiv q).\sigma & \doteq & p.\sigma \equiv q.\sigma & (\equiv \text{ en } \mathsf{Pred}) \\
(p \Rightarrow q).\sigma & \doteq & p.\sigma \Rightarrow q.\sigma & (\Rightarrow \text{ en } \mathsf{Pred})
\end{array}$$

Las siguientes operaciones de Pred devuelven elementos en Bool:

$$\begin{array}{rcll}
p \subseteq q & \doteq & \langle \forall \sigma : p.\sigma : q.\sigma \rangle & (\text{orden en } \mathsf{Pred}) \\
p = q & \doteq & \langle \forall \sigma :: (p \equiv q).\sigma \rangle & (\text{igualdad en } \mathsf{Pred})
\end{array}$$

Cabe remarcar que la igualdad entre predicados $_ = _ : \mathsf{Pred} \to \mathsf{Pred} \to \mathsf{Bool}$ es un operador binario distinto que la equivalencia $_ \equiv _ : \mathsf{Pred} \to \mathsf{Pred} \to \mathsf{Pred}$. El primero indica la igualdad entre elementos de Pred, formulada en base a la antisimetría del orden en este reticulado. El segundo denota una operación que devuelve como resultado elementos del mismo reticulado. Este último suele identificarse con la doble implicación (si y solo si) entre predicados (devuelve un predicado) aunque no debe confundirse con la doble implicación en Bool (devuelve un elemento en Bool).

Como el reticulado Pred es completo, podemos definir las operaciones de ínfimo y supremo de conjuntos. Estas serán denotadas como expresiones cuantificadas sobre conjuntos indexados. Sean $I \subseteq \Gamma$ conjuntos cualesquiera (I será denominado *conjunto de índices*) y $P : \Gamma \to \mathsf{Pred}$, definiremos el ínfimo y supremo

[6] La propiedad de atomicidad no se hereda necesariamente por extensión puntual. Un ejemplo de este hecho puede encontrarse en [BAW98, ejercicio 2.6].

de los conjuntos en $\{P.i \mid i \in I\}$ como:

$$\langle \bigcap i : I.i : P.i \rangle.\sigma \doteq \langle \forall i : I.i : P.i.\sigma \rangle \qquad \text{(ínfimo en Pred)}$$
$$\langle \bigcup i : I.i : P.i \rangle.\sigma \doteq \langle \exists i : I.i : P.i.\sigma \rangle \qquad \text{(supremo en Pred)}$$

Claramente, el reticulado Pred es isomorfo al de conjuntos de estados: el orden $\subseteq$ se identifica a la inclusión de conjuntos, el predicado false al conjunto vacío, true al conjunto universal, el ínfimo binario $\cap$ a la intersección y el supremo binario $\cup$ a la unión entre conjuntos. Por este motivo, en este trabajo hablaremos indistintamente sobre predicados o conjuntos de estados, definiendo también a Pred como las partes de Σ:

$$\mathsf{Pred} \doteq \wp.\Sigma \ .$$

En este mismo sentido, denotaremos predicados como funciones $\langle \lambda\sigma : \Sigma \bullet E \rangle$ o como conjuntos $\{\sigma : \Sigma \mid E\}$ con E una término booleano sobre σ.

A continuación introduciremos la terminología que utilizaremos de aquí en adelante.

Terminología
Dados p, q : Pred.

Cuando $p.\sigma = \mathsf{T}$ diremos que

- σ *cumple* p.
- p es *verdadero* en σ,
- p *describe* a σ,
- σ *satisface* a p.

Cuando $p.\sigma = \mathsf{F}$ diremos que

- p es *falso* en σ,
- σ *no satisface* a p.

Cuando p es verdadero para todo estado $\sigma \in \Sigma$, diremos que

- p es *válida*.

  Formalmente $\langle \forall \sigma : \sigma \in \Sigma : p.\sigma \rangle$.

Cuando p es verdadero para algún estado $\sigma \in \Sigma$, diremos que

- p es *satisfactible*.

  Formalmente $\langle \exists \sigma : \sigma \in \Sigma : p.\sigma \rangle$.

Cuando p es falso para todo estado $\sigma \in \Sigma$, diremos que

- p es *no satisfactible*.

  Formalmente $\langle \forall \sigma : \sigma \in \Sigma : \neg p.\sigma \rangle$.

Cuando $p \subseteq q$, diremos que

- p es *más fuerte* que q,
- q es *más débil* que p.

Cuando $p = q$, diremos que

– p es *equivalente* a q. □

1.2.1 Sintaxis

Como mencionamos en la sección 1.1 en nuestro trabajo expresaremos la formalización a un nivel semántico utilizando el enfoque denominado *shallow embedding*. Este nivel de desarrollo semántico pudo verse claramente al definir predicados como funciones de los estados abstractos al reticulado de valores de verdad. De todas maneras, resulta engorroso y poco intuitivo escribir un predicado como una función, por lo cual este enfoque permite el uso de las expresiones usuales que denotan esta semántica a la manera de abreviaciones, esto es, como objetos de carácter sintáctico fuera de la lógica. De esta forma las expresiones booleanas usuales abreviarán directamente los predicados vistos anteriormente. Por ejemplo, si el espacio de estados son las funciones entre variables y valores en $\mathbb{Z}$ (dotados de la aritmética usual), la expresión booleana $n \geqslant 0 \wedge a < n$ abreviará el predicado $\langle \lambda\sigma \bullet \sigma.n \geqslant 0 \wedge \sigma.a < \sigma.n \rangle$ (o lo que es lo mismo $\langle \lambda\sigma \bullet \sigma.n \geqslant 0 \rangle \cap \langle \lambda\sigma \bullet \sigma.a < \sigma.n \rangle$). De aquí en adelante estas abreviaturas las señalaremos con el símbolo "$\triangleq$":

$$n \geqslant 0 \wedge a < n \triangleq \langle \lambda\sigma \bullet \sigma.n \geqslant 0 \wedge \sigma.a < \sigma.n \rangle \ .$$

Con las *expresiones universales* y *existenciales* procederemos de forma similar. Por ejemplo, dado el espacio de estados $\Sigma \doteq \mathsf{Var} \to \mathbb{Z}$, la expresión $\langle \forall x \ : x + y \geqslant 0 : \ x < y + 1 \rangle$ representará el ínfimo $\langle \bigcap i \ : I.i : \ R.i \Rightarrow P.i \rangle$, con $I \doteq \mathbb{Z}$, $R.i \doteq \langle \lambda\sigma \bullet i + \sigma.y \geqslant 0 \rangle$ y $P.i \doteq \langle \lambda\sigma \bullet i < \sigma.y + 1 \rangle$.

De forma general, si las expresiones r y p abrevian los predicados r y p respectivamente, entonces

$$\langle \forall v_1, \cdots, v_k \ : \mathsf{r} : \ \mathsf{p} \rangle \triangleq \langle \bigcap i_1, \cdots, i_k \ : I.i_1.\cdots.i_k : \ R.i_1.\cdots.i_k \Rightarrow P.i_1.\cdots.i_k \rangle$$

con $I = \mathbb{Z}^k$,
$R.i_1.\cdots.i_k.\sigma \doteq r.(\sigma[v_1 \mapsto i_1, \cdots, v_k \mapsto i_k])$ y
$P.i_1.\cdots.i_k.\sigma \doteq p.(\sigma[v_1 \mapsto i_1, \cdots, v_k \mapsto i_k])$,

donde $\sigma[v_1 \mapsto i_1, \cdots, v_k \mapsto i_k]$ es la actualización usual de la función σ.

De manera análoga se representarán las expresiones cuantificadas existenciales:

$$\langle \exists v_1, \cdots, v_k \ : \mathsf{r} : \ \mathsf{p} \rangle \triangleq \langle \bigcup i_1, \cdots, i_k \ : I.i_1.\cdots.i_k : \ R.i_1.\cdots.i_k \cap P.i_1.\cdots.i_k \rangle$$

1.3 Sentencias de programa

Como ya mencionamos, los programas imperativos trabajan alterando sucesivamente la memoria. Esto es, en cada paso de ejecución de una sentencia de programa, la memoria (representada por un estado en particular), es modificada produciendo un estado nuevo. En esta sección nos concentraremos en el modelado de las ejecuciones individuales de estas sentencias. En la sección siguiente veremos cómo modelar programas en general.

Existen varias formas de denotar sentencias en un nivel semántico. Si las sentencias son deterministas una forma usual es denotarlas como funciones desde un estado inicial a otro final. Un problema que surge es el de significar la no terminación: pueden existir estados iniciales desde los cuales una sentencia no termina, por lo cual la función que la denota no está definida para todos los puntos del espacio de estados. Ante este problema existen dos opciones: utilizar funciones parciales (funciones sobre un dominio más pequeño que el considerado) o hacerlas totales con tipo $\Sigma \to \Sigma \cup \{\bot\}$ donde la no terminación es indicada explícitamente con el estado final especial denominado *bottom* ($\bot$) [MM04, sección 5.1]. Ambos enfoques poseen sus desventajas. La primera opción introduce funciones que no pueden ser aplicadas libremente a un argumento por más que posea el tipo correcto. El segundo método tiene la ventaja de utilizar funciones totales, pero a costa de destruir la homogeneidad del espacio de estados.

Otra forma de denotar sentencias es utilizando relaciones. De esta manera, el conjunto *sentencias* serán las relaciones entre estados:

$$\mathsf{Sent} \;\doteq\; \Sigma \to \Sigma \to \mathsf{Bool}$$

lo cual permitirá homogeneizar el espacio de estados. Además podremos denotar sentencias no deterministas: una sentencia de programa podrá ser especificada de forma tal que desde un estado inicial haya un conjunto de estados finales alcanzables. El incluir esta posibilidad permite modelar programas con elecciones de cambio de estado externas al mismo (por ejemplo por intervención del usuario, por decisión de un scheduler, etc.).

En este contexto, una sentencia s será determinista cuando la cardinalidad del conjunto $s.\sigma$ es menor o igual a uno ($\mathsf{Card}.(s.\sigma) \leqslant 1$) para todo σ. Cuando esta cardinalidad es igual a cero para todo σ la sentencia no termina. Notar además que Sent puede ser visto como la extensión puntual de los predicados a los estados, esto es $\mathsf{Sent} = \Sigma \to \mathsf{Pred}$. Por lo tanto Sent es un reticulado booleano y completo atómico ya que hereda estas propiedades de Pred y además puede demostrarse que es atómico [BAW98, teorema 9.1].

Nuestra decisión de utilizar relaciones para formalizar la semántica de las sentencias no está libre de problemas; si estas son no deterministas es imposible significar las computaciones individuales que no terminan (para el caso determinista la única computación se representa simplemente con la relación vacía). De todas maneras en esta exposición nos va a interesar significar solo los estados alcanzables desde un estado inicial independientemente de cualquier elección no determinista que pueda hacer abortar el programa.

Como vimos, una sentencia toma un estado inicial y devuelve un conjunto de estados que representan las posibles elecciones no deterministas de la sentencia. Además, si este conjunto es vacío querrá significar que la sentencia no termina para ese estado en particular. En este sentido definiremos la siguiente terminología.

Terminología
Dados $s : \mathsf{Sent}$ y $\sigma : \Sigma$

- $s.\sigma$ será denominado *conjunto de estados alcanzables* por s a partir de σ.
- Cuando $s.\sigma = \mathsf{false}$ diremos que la sentencia *s no termina desde σ*.
- Cuando $\langle \forall \sigma \; :: \; s.\sigma = \mathsf{false} \rangle$ diremos que la sentencia *s no termina*.

– Cuando $\langle \forall \sigma \, :: \, \neg(s.\sigma = \mathsf{false}) \rangle$ diremos que la sentencia s es *total*.

– Cuando $\langle \forall \sigma \, :: \, \mathsf{Card}.(s.\sigma) \leqslant 1 \rangle$ diremos que la sentencia es *determinista*.

□

Ejemplo 1.1
A continuación presentaremos dos constructores de sentencias que utilizaremos más adelante. Dado el espacio de funciones entre estados definido como $\mathsf{Fun} \doteq \Sigma \to \Sigma$, definiremos las siguientes funciones:

$$\begin{array}{|l} \langle _ \rangle : \mathsf{Fun} \to \mathsf{Sent} \\ [_] : \mathsf{Pred} \to \mathsf{Sent} \\ \hline \langle f \rangle.\sigma.\sigma' \doteq f.\sigma = \sigma' \\ [b].\sigma.\sigma' \doteq b.\sigma \wedge \sigma = \sigma' \ . \end{array}$$

A la primera ($\langle _ \rangle$) la denominaremos *actualización funcional* y suele utilizarse para modelar asignaciones. A la segunda ($[_]$) la denominaremos *suposición*, la cual relaciona un estado final idéntico al inicial siempre y cuando el predicado sea verdadero en el mismo. Si el predicado es falso en un estado σ la sentencia no termina desde σ. Ambas sentencias son deterministas.

Vamos a definir también el operador sobre sentencias denominado *composición secuencial* como composición de relaciones:

$$\begin{array}{|l} _ ; _ : \mathsf{Sent} \to \mathsf{Sent} \to \mathsf{Sent} \\ \hline (s_1; s_2).\sigma.\sigma' \doteq \langle \exists \sigma'' \, :: \, s_1.\sigma.\sigma'' \wedge s_2.\sigma''.\sigma' \rangle \ . \end{array}$$

Notar que si ambos parámetros del operador son deterministas el resultado también lo es.

Utilizando estos constructores básicos podemos desarrollar sentencias más complejas. Un programa que utilizaremos más adelante es la composición secuencial de una suposición con una actualización funcional $[b]; \langle f \rangle$. Su definición puede derivarse a partir de las de los constructores básicos:

$$\begin{array}{ll} & ([b]; \langle f \rangle).\sigma.\sigma' \\ \equiv & \{ \text{ Definición de composición secuencial } \} \\ & \langle \exists \sigma'' \, :: \, [b].\sigma.\sigma'' \wedge \langle f \rangle.\sigma''.\sigma' \rangle \\ \equiv & \{ \text{ Definición de suposición y actualización funcional } \} \\ & \langle \exists \sigma'' \, :: \, b.\sigma \wedge \sigma = \sigma'' \wedge f.\sigma'' = \sigma' \rangle \\ \equiv & \{ \text{ Rango unitario } \} \\ & b.\sigma \wedge f.\sigma = \sigma' \ . \end{array}$$

Por lo tanto

$$([b]; \langle f \rangle).\sigma.\sigma' \doteq b.\sigma \wedge f.\sigma = \sigma' \ .$$

Esta construcción opera sobre un estado inicial aplicando la actualización funcional, siempre y cuando el estado inicial satisfaga el predicado de la suposición. Si un estado no la satisface la sentencia no termina desde ese estado.

Puede demostrarse que cualquier sentencia s determinista puede escribirse de la forma $[b]; \langle f \rangle$, tomando a b el dominio de la relación y a f la función que asigna a cada estado su relacionado. Utilizando el *operador de elección* ϵ esto último puede escribirse como $f.\sigma \doteq \epsilon.(s.\sigma)$. □

1.3.1 Sintaxis

Siguiendo el estilo de formalización *shallow embedding* (sección 1.1, pág. 1) introduciremos algunas construcciones del lenguaje.

La *suposición* a nivel sintáctico será definida como sigue. Si b es un predicado y B una expresión booleana, tal que $\mathsf{B} \triangleq b$ entonces la sentencia de suposición $[\mathsf{B}]$ será definida como:

$$[\mathsf{B}] \triangleq [b] \ .$$

La sentencia denominada usualmente como *asignación múltiple* será una abreviación de la actualización funcional. Por ejemplo, si $\Sigma \doteq \mathsf{Var} \to \mathbb{Z}$, entonces

$$x, y := y + 1, n \triangleq \langle f \rangle$$

con

$$f.\sigma = \sigma[x \mapsto \sigma.y + 1, y \mapsto \sigma.n] \ .$$

Notar que si el conjunto de variables Var es finito resulta totalmente válido expresar las actualizaciones funcionales de esta manera siempre que el poder expresivo de las expresiones lo permitan. Esto es, si el conjunto de variables es $\mathsf{Var} \doteq \{v_1, \cdots, v_n\}$ y existen expresiones $e_1, \cdots, e_n$ tal que e_i abrevia la función $\langle \lambda\sigma \bullet (f.\sigma).v_i \rangle$ entonces

$$v_1, \cdots, v_n := e_1, \cdots, e_n \triangleq \langle f \rangle \ .$$

También utilizaremos como expresiones de sentencias la siguiente construcción que denominaremos *asignación guardada*:

$$\square\ \mathsf{B} \mapsto v_1, \cdots, v_k := e_1, \cdots, e_k \ \triangleq \ [b]; \langle f \rangle$$

con

$$\mathsf{B} \triangleq b \qquad \text{y} \qquad v_1, \cdots, v_k := e_1, \cdots, e_k \triangleq \langle f \rangle \ .$$

Otras abreviaciones que utilizaremos serán

$$\begin{array}{rcll} \textbf{skip} & \triangleq & \langle \mathsf{Id} \rangle & \text{con } \mathsf{Id} \text{ la función identidad} \\ \textbf{abort} & \triangleq & [\mathsf{false}] & \\ \square\ \mathsf{B} \mapsto \textbf{skip} & \triangleq & [b] & \text{con } \mathsf{B} \triangleq b \ . \end{array}$$

Las dos primeras denotan las sentencias *skip* y *abort* usuales. La última se utiliza para escribir una suposición en la forma de sentencia guardada.

1.4 Transformadores de predicados

Un transformador de predicados [DS90, BAW98] será una función de Pred en si mismo. La noción de transformador de predicados nos permitirá, a partir de una programa en Sent, definir distintas semánticas. A continuación las desarrollaremos.

1.4.1 Strongest postcondition

La semántica de *strongest postcondition* será definida a partir del transformador de predicados sp. El mismo toma una sentencia y un predicado y devuelve el conjunto de estados que están relacionados, por medio de la sentencia, con alguno contenido en el predicado parámetro:

$$\text{sp} : \mathsf{Sent} \to \mathsf{Pred} \to \mathsf{Pred}$$
$$\text{sp}.s.p.\sigma' \doteq \langle \exists \sigma : p.\sigma : s.\sigma.\sigma' \rangle$$

Otra forma de ver este transformador es pensar el predicado sp.$s.p$ como el conjunto de estados alcanzables desde los estados en p por la sentencia s:

$$\text{sp}.s.p \doteq \langle \bigcup \sigma : p.\sigma : s.\sigma \rangle \ .$$

Ambas definiciones son equivalentes ya que

$$\langle \bigcup \sigma : p.\sigma : s.\sigma \rangle.\sigma' \equiv \langle \exists \sigma : p.\sigma : s.\sigma.\sigma' \rangle$$

por la definición de supremo en la sección 1.2.

Ejemplo 1.2
Veamos el resultado de aplicar este transformador de predicados a la sentencia $[b]$:

$$\begin{aligned}
& \text{sp}.[b].p.\sigma' \\
\equiv & \ \{ \text{ Definición de sp } \} \\
& \langle \exists \sigma : p.\sigma : [b].\sigma.\sigma' \rangle \\
\equiv & \ \{ \text{ Definición de suposición } \} \\
& \langle \exists \sigma : p.\sigma : b.\sigma \wedge \sigma = \sigma' \rangle \\
\equiv & \ \{ \text{ Rango unitario } \} \\
& p.\sigma' \wedge b.\sigma' \ .
\end{aligned}$$

Por lo tanto $\text{sp}.[b].p.\sigma' \doteq p.\sigma' \wedge b.\sigma'$.

Utilizando la definición de conjunción de predicados (sección 1.2), esta equivalencia puede expresarse de forma simple como

$$\text{sp}.[b].p = p \cap b \ ,$$

la cual expresa que los estados alcanzables desde el predicado p por la ejecución de la sentencia $[b]$ deben cumplir el predicado b. Si esto no sucede la sentencia no termina. □

Ejemplo 1.3
Veamos el resultado de aplicar este transformador de predicados a la sentencia $[b];\langle f \rangle$:

$$\begin{aligned}
& \text{sp}.([b];\langle f \rangle).p.\sigma' \\
\equiv & \ \{ \text{ Definición de sp } \} \\
& \langle \exists \sigma : p.\sigma : ([b];\langle f \rangle).\sigma.\sigma' \rangle
\end{aligned}$$

$\equiv$ { Definición de $[b];\langle f \rangle$ en ejemplo 1.1 }
$\langle \exists\, \sigma \; : p.\sigma : \; b.\sigma \wedge f.\sigma = \sigma' \rangle$
$\equiv$ { Intercambio }
$\langle \exists\, \sigma \; : p.\sigma \wedge b.\sigma : \; f.\sigma = \sigma' \rangle$
$\equiv$ { Conjunción de predicados }
$\langle \exists\, \sigma \; : (p \cap b).\sigma : \; f.\sigma = \sigma' \rangle$.

Este último resultado expresa que los estados alcanzables desde el predicado p por la ejecución de $[b];\langle f \rangle$ son la imagen de la función f sobre el conjunto de estados $p \cap b$. □

El trasformador sp posee las siguientes propiedades sobre las sentencias:

Propiedades 1.4 (sp en sentencias)

$$
\begin{aligned}
\text{sp}.[b].p &= b \cap p \\
\text{sp}.\langle f \rangle.p &= p \circ f^{-1} \text{ siempre que } f \text{ tenga inversa} \\
\text{sp}.(s_1; s_2).p &= \text{sp}.s_2.(\text{sp}.s_1.p)
\end{aligned}
$$

□

La demostración es directa a partir de la definición de sp.

1.4.2 Weakest liberal precondition

Otra semántica usual es la definida a partir del transformador *weakest liberal precondition*. El transformador toma una sentencia y un predicado, devolviendo el conjunto de estados tales que, si la sentencia termina, lo hace a un estado perteneciente al predicado parámetro:

$$
\begin{array}{|l}
\text{wlp} : \mathsf{Sent} \to \mathsf{Pred} \to \mathsf{Pred} \\
\hline
\text{wlp}.s.q.\sigma \doteq \langle \forall\, \sigma' \; : s.\sigma.\sigma' : \; q.\sigma' \rangle
\end{array}
$$

Una formulación equivalente, aplicando la definición de $\subseteq$ (sección 1.2), es:

$$\text{wlp}.s.q.\sigma \doteq s.\sigma \subseteq q$$

Notar, a partir de esta última definición, que si para un estado particular σ la sentencia s no termina ($s.\sigma$ es el conjunto vacío), este pertenece al resultado del transformador (o sea a $\text{wlp}.s.q$). Con ello se observa claramente que el transformador wlp no significa la terminación de la sentencia[7].

Ejemplo 1.5
Veamos el resultado de aplicar el transformador wlp al programa $[b];\langle f \rangle$:

$\text{wlp}.([b];\langle f \rangle).q.\sigma$
$\equiv$ { Definición de wlp }
$\langle \forall\, \sigma' \; : ([b];\langle f \rangle).\sigma.\sigma' : \; q.\sigma' \rangle$

[7] Con el fin de capturar la no terminación de los programas, en la literatura [DS90] se define además el transformador *weakest precondition*. En este trabajo no lo utilizaremos ya que la terminación no será una propiedad que nos interese al momento de refinar programas concurrentes, como veremos más adelante.

$$
\begin{aligned}
&\equiv \{\ \text{Definición de } [b];\langle f\rangle \text{ en ejemplo } 1.1\ \}\\
&\quad \langle\, \forall\, \sigma'\ : b.\sigma \wedge f.\sigma = \sigma' :\ q.\sigma'\,\rangle\\
&\equiv \{\ \text{Intercambio}\ \}\\
&\quad \langle\, \forall\, \sigma'\ : f.\sigma = \sigma' :\ b.\sigma \Rightarrow q.\sigma'\,\rangle\\
&\equiv \{\ \text{Rango unitario}\ \}\\
&\quad b.\sigma \Rightarrow q.(f.\sigma)\\
&\equiv \{\ \text{Composición de funciones}\ \}\\
&\quad b.\sigma \Rightarrow (q \circ f).\sigma\\
&\equiv \{\ \text{Definición de} \Rightarrow\ \}\\
&\quad (b \Rightarrow q \circ f).\sigma\ \ .
\end{aligned}
$$

Por lo tanto:

$$\text{wlp}.([b];\langle f\rangle).q\ \doteq\ b \Rightarrow q \circ f\ \ .$$

Este último resultado expresa que los estados en que la sentencia $[b];\langle f\rangle$ si termina lo hace en q, son aquellos que no pertenecen a b o que aplicados a f pertenecen a q. □

El resultado puede también deducirse de las siguientes propiedades particulares del transformador sobre las sentencias:

Propiedades 1.6 (wlp en sentencias)

$$
\begin{aligned}
\text{wlp}.[b].q &= b \Rightarrow q\\
\text{wlp}.\langle f\rangle.q &= q \circ f\\
\text{wlp}.(s_1;s_2).q &= \text{wlp}.s_1.(\text{wlp}.s_2.q)
\end{aligned}
$$

□

La demostración es directa a partir de la definición de wlp.

1.4.3 Propiedades de los transformadores

Los transformadores de predicados en general, como funciones de predicados en predicados, pueden ser organizados según cierta jerarquía de propiedades [DS90, capítulo 6] que se preservan como homomorfismos entre reticulados [BAW98, capítulo 16]. Presentaremos las mismas teniendo en mente el reticulado Pred aunque también pueden ser generalizadas a cualquier reticulado completo e incluso a otros tipos de conjuntos parcialmente ordenados (veremos esto más detalladamente en el capítulo 2).

Definición 1.7 (Monotonía)
Un transformador de predicados $F : \mathsf{Pred} \to \mathsf{Pred}$ es *monótono* si preserva la relación $\subseteq$. Esto es, para todo par de predicados $X, Y : \mathsf{Pred}$ se cumple:

$$X \subseteq Y \Rightarrow F.X \subseteq F.Y\ \ .$$

□

Definición 1.8 (Juntividad)
Un transformador de predicados es *estricto* si preserva false y es *no interrumpible*[8] si preserva true. Además es *positivamente conjuntivo* si preserva ínfimos no vacíos de predicados y es *positivamente disyuntivo* si preserva supremos no vacíos de predicados.

8 En inglés *terminating* o *nonaborting*.

Formalmente, dado $F : \mathsf{Pred} \to \mathsf{Pred}$,

- F es estricto si $F.\mathsf{false} = \mathsf{false}$.
- F es no interrumpible si $F.\mathsf{true} = \mathsf{true}$.
- F es positivamente conjuntivo si, para todo conjunto de índices $I \subseteq \Gamma$ con $I \neq \varnothing$ y para todo $P : \Gamma \to \mathsf{Pred}$, se cumple:

  $$F.\langle \bigcap i \ : I.i : \ P.i \rangle = \langle \bigcap i \ : I.i : \ F.(P.i) \rangle \ .$$

  Esta propiedad indica que un transformador distribuye con respecto a conjunciones no vacías de predicados.
- F es positivamente disyuntivo si, para todo conjunto de índices $I \subseteq \Gamma$ con $I \neq \varnothing$ y para todo $P : \Gamma \to \mathsf{Pred}$, se cumple:

  $$F.\langle \bigcup i \ : I.i : \ P.i \rangle = \langle \bigcup i \ : I.i : \ F.(P.i) \rangle \ .$$

  Esta propiedad indica que un transformador distribuye con respecto a disyunciones no vacías de predicados.
- Un transformador es *universalmente conjuntivo* si preserva ínfimos arbitrarios de predicados (es no interrumpible y positivamente conjuntivo).
- Un transformador es *universalmente disyuntivo* si preserva supremos arbitrarios de predicados (es estricto y positivamente disyuntivo). □

Otra propiedad interesante es la que permite la noción de continuidad de transformadores. Para presentarla definiremos antes el concepto de *conjunto dirigido* y *conjunto codirigido*.

Definición 1.9 (Conjunto dirigido)
Un subconjunto C de un poset $(P, \sqsubseteq)$ es *dirigido* si todo par de elementos en C tienen cota superior en el mismo:

$$\langle \forall a, b \ : a, b \in C : \ \langle \exists c \ : c \in C : \ a \sqsubseteq c \wedge b \sqsubseteq c \rangle \rangle \ .$$

De forma dual, un subconjunto C de un poset $(P, \sqsubseteq)$ es *codirigido* si todo par de elementos en C tienen cota inferior en el mismo:

$$\langle \forall a, b \ : a, b \in C : \ \langle \exists c \ : c \in C : \ c \sqsubseteq a \wedge c \sqsubseteq b \rangle \rangle \ .$$ □

Ejemplos de conjuntos dirigidos son las cadenas, las partes de un conjunto $\wp.X$ y el conjunto vacío. A partir de esta definición introduciremos el concepto de continuidad de un transformador:

Definición 1.10 (Continuidad)
Un transformador de predicados es *or-continuo* (*and-continuo*) si es disyuntivo (conjuntivo) sobre conjuntos dirigidos (codirigidos) no vacíos. Formalmente, dado $F : \mathsf{Pred} \to \mathsf{Pred}$,

- F es or-continuo si, para todo conjunto de índices $I \subseteq \Gamma$ con $I \neq \varnothing$ y para todo $P : \Gamma \to \mathsf{Pred}$ tal que $\{P.i \mid i \in I\}$ es dirigido, se cumple:

  $$F.\langle \bigcup i \ : I.i : \ P.i \rangle = \langle \bigcup i \ : I.i : \ F.(P.i) \rangle \ .$$

- F es and-continuo si, para todo conjunto de índices $I \subseteq \Gamma$ con $I \neq \emptyset$ y para todo $P : \Gamma \to \mathsf{Pred}$ tal que $\{P.i \mid i \in I\}$ es codirigido, se cumple:

$$F.\langle \bigcap i \ : I.i : \ P.i \rangle = \langle \bigcap i \ : I.i : \ F.(P.i) \rangle \ . \qquad \square$$

Como mencionamos, estas propiedades pueden ser ordenadas según cierta jerarquía:

Teorema 1.11 (Jerarquía de propiedades)
Dado un transformador de predicados $F : \mathsf{Pred} \to \mathsf{Pred}$ se cumplen las siguientes implicaciones:

F es universalmente conjuntivo
$\Rightarrow F$ es positivamente conjuntivo
$\Rightarrow F$ es and-continuo
$\Rightarrow F$ es monótono

y de forma dual

F es universalmente disyuntivo
$\Rightarrow F$ es positivamente disyuntivo
$\Rightarrow F$ es or-continuo
$\Rightarrow F$ es monótono $\square$

Veamos ahora, cuales de estas propiedades son satisfechas por los transformadores sp y wlp:

Propiedades 1.12 (Transformadores sp y wlp)
Dado $s : \mathsf{Sent}$

1. sp.s es universalmente disyuntivo. A partir de esta propiedad se puede deducir que es estricto (sp.s.$\mathsf{false} = \mathsf{false}$), monótono y or-continuo.

2. wlp.s es universalmente conjuntivo. A partir de esta propiedad se puede deducir que es es no interrumpible (wlp.s.$\mathsf{true} = \mathsf{true}$), monótono y and-continuo[9].

3. Si s es determinista, sp.s es positivamente conjuntivo.

4. Si s es determinista, wlp.s es positivamente disyuntivo.

5. Si s es total entonces wlp.s es estricto (wlp.s.$\mathsf{false} = \mathsf{false}$).

6. Si s es determinista entonces

$$\text{wlp}.s.(P \Rightarrow Q) \ = \ \text{wlp}.s.P \Rightarrow \text{wlp}.s.Q$$

(wlp distribuye con la implicación).

7. Si s es determinista y total entonces para todo predicado P se cumple

$$\text{wlp}.s.\neg P \ = \ \neg \, \text{wlp}.s.P \ \ .$$

[9] Esto confirma lo mencionado al comienzo de esta sección: esta semántica no discrimina la no terminación ya que **abort** cumple esta propiedad igual que cualquier otra sentencia.

Demostración
La demostración de las propiedades 1 a 4 pueden encontrarse en [DS90, BAW98]. A continuación demostraremos las siguientes.

5. Utilizando que $s.\sigma$ es no vacío para cualquier σ ($s.\sigma$ termina) tenemos:

$$
\begin{array}{ll}
& \mathrm{wlp}.s.\mathsf{false}.\sigma \\
\equiv & \{ \text{ Definición de wlp } \} \\
& \langle \forall \sigma' : s.\sigma.\sigma' : \mathsf{false}.\sigma' \rangle \\
\equiv & \{ \text{ Definición de } \mathsf{false} \} \\
& \langle \forall \sigma' : s.\sigma.\sigma' : \mathsf{F} \rangle \\
\equiv & \{ s.\sigma \text{ es no vacío y término constante } \} \\
& \mathsf{F}
\end{array}
$$

6. Consideremos los casos $s.\sigma = \mathsf{false}$ y $s.\sigma = \{\tilde{\sigma}\}$ ($s.\sigma$ no termina o termina de forma determinista).

Caso ($s.\sigma = \mathsf{false}$)

$$
\begin{array}{ll}
& (\mathrm{wlp}.s.P \Rightarrow \mathrm{wlp}.s.Q).\sigma \\
\equiv & \{ \text{ Lógica de predicados, definición de wlp } \} \\
& \langle \forall \sigma' : s.\sigma.\sigma' : P.\sigma' \rangle \Rightarrow \langle \forall \sigma' : s.\sigma.\sigma' : Q.\sigma' \rangle \\
\equiv & \{ \text{ Rango vacío por caso } \} \\
& \mathsf{T} \Rightarrow \mathsf{T} \\
\equiv & \{ \text{ Lógica } \} \\
& \mathsf{T} \\
\equiv & \{ \text{ Rango vacío por caso } \} \\
& \langle \forall \sigma' : s.\sigma.\sigma' : (P \Rightarrow Q).\sigma' \rangle \\
\equiv & \{ \text{ Definición de wlp } \} \\
& \mathrm{wlp}.s.(P \Rightarrow Q).\sigma
\end{array}
$$

Caso ($s.\sigma = \{\tilde{\sigma}\}$)

$$
\begin{array}{ll}
& (\mathrm{wlp}.s.P \Rightarrow \mathrm{wlp}.s.Q).\sigma \\
\equiv & \{ \text{ Lógica de predicados, definición de wlp } \} \\
& \langle \forall \sigma' : s.\sigma.\sigma' : P.\sigma' \rangle \Rightarrow \langle \forall \sigma' : s.\sigma.\sigma' : Q.\sigma' \rangle \\
\equiv & \{ \text{ Rango unitario por caso } \} \\
& P.\tilde{\sigma} \Rightarrow Q.\tilde{\sigma} \\
\equiv & \{ \text{ Lógica de predicados } \} \\
& (P \Rightarrow Q).\tilde{\sigma} \\
\equiv & \{ \text{ Rango unitario por caso } \} \\
& \langle \forall \sigma' : s.\sigma.\sigma' : (P \Rightarrow Q).\sigma' \rangle \\
\equiv & \{ \text{ Definición de wlp } \} \\
& \mathrm{wlp}.s.(P \Rightarrow Q).\sigma
\end{array}
$$

7. Demostraremos esta propiedad utilizando las anteriores:

$$
\begin{array}{ll}
& \text{wlp}.s.\neg P \\
= & \{ \text{ Lógica de predicados } \} \\
& \text{wlp}.s.(P \Rightarrow \mathsf{false}) \\
= & \{ \ s \text{ es determinista y propiedad } 6 \ \} \\
& \text{wlp}.s.P \Rightarrow \text{wlp}.s.\mathsf{false} \\
= & \{ \ s \text{ termina para cualquier estado y propiedad } 5 \ \} \\
& \text{wlp}.s.P \Rightarrow \mathsf{false} \\
= & \{ \text{ Lógica de predicados } \} \\
& \neg\, \text{wlp}.s.P
\end{array}
$$

□

A continuación presentaremos algunas propiedades que relacionan los transformadores sp y wlp. La siguiente muestra que la 3-upla de Hoare $\{p\}\ s\ \{q\}$ puede ser interpretada con la fórmula clásica $p \subseteq \text{wlp}.s.q$ o utilizando el transformador sp:

Propiedad 1.13 (Equivalencia sp wlp)
Para toda sentencia s y predicados p q se cumple:

$$p \subseteq \text{wlp}.s.q \;\equiv\; \text{sp}.s.p \subseteq q \ .$$

□

Otra propiedad interesante es la que sugiere cierta noción de dualidad entre los transformadores.

Propiedad 1.14 (Dualidad entre sp y wlp)
Dada la sentencia s, definimos como s^{-1} a su relación inversa[10]. Entonces se cumple:

$$\text{sp}.s.p \;=\; \neg(\text{wlp}.s^{-1}.\neg p) \ .$$

□

La demostración es directa de la definición de los transformadores.

1.5 Sistemas de transiciones

En la sección anterior vimos como utilizando un modelo relacional es posible representar las sentencias individuales de un programa. Esto nos permitió denotar sentencias simples como asignaciones guardadas. En esta sección extenderemos el formalismo para poder representar programas más complejos que contengan bucles y saltos no estructurados[11].

Sea $\mathcal{S}$ un conjunto finito de sentencias ($\mathcal{S} \subseteq \mathsf{Sent}$). Sea $\mathcal{L}$ un conjunto finito que representa los posibles valores del contador de programa al cual denominaremos *conjunto de locaciones*. Sea $\mathsf{Tran}_{\mathcal{L},\mathcal{S}}$ las relaciones entre dos locaciones y sentencias en $\mathcal{S}$:

$$\mathsf{Tran}_{\mathcal{L},\mathcal{S}} \;\doteq\; \mathcal{L} \to \mathcal{S} \to \mathcal{L} \to \mathsf{Bool}$$

Una relación $\mathcal{T}$: $\mathsf{Tran}_{\mathcal{L},\mathcal{S}}$ puede ser vista como un conjunto de 3-uplas en $\mathcal{L} \times \mathcal{S} \times \mathcal{L}$ a las que llamaremos *transiciones*. La primera y tercera coordenadas de una transición serán denominadas *locación de salida* y *locación de llegada*

[10] La inversa de una relación r se define de forma usual como $r^{-1}.a.b \equiv r.b.a$.

[11] También denominados sentencias goto.

respectivamente. Dada una transición con locación de salida l y locación de llegada l' diremos que es una *transición saliente* desde l y que es una *transición entrante* hacia l'.

Sea $\mathsf{Pred}_{\mathcal{L}}$ la extensión puntual de los predicados a las locaciones:

$$\mathsf{Pred}_{\mathcal{L}} \doteq \mathcal{L} \to \mathsf{Pred} \ .$$

Un elemento de tipo $\mathsf{Pred}_{\mathcal{L}}$ representa un predicado para cada locación o más concretamente, define un conjunto de estados para cada locación. Llamaremos *predicado de un sistema de transiciones* a un elemento de $\mathsf{Pred}_{\mathcal{L}}$, aunque también los nombraremos predicados cuando la distinción con los elementos de Pred se sobreentienda a partir del contexto. Diremos además que un par *estado/locación* pertenece a un predicado de un sistemas de transiciones cuando el estado pertenezca a la valuación del predicado en la locación. A los pares estado/locación lo denominaremos *configuraciones*.

Por último definiremos el *conjunto de configuraciones iniciales* $\Theta : \mathsf{Pred}_{\mathcal{L}}$. Generalmente, para programas simples, Θ evaluado en las locaciones $(\Theta.l)$ es satisfactible solo en una de ellas, la cual representa el punto de ejecución inicial del programa. Llamaremos *locaciones iniciales* a las locaciones $l \in \mathcal{L}$ tales que $\Theta.l$ es satisfactible. Como veremos luego, la caracterización de configuraciones iniciales omitiendo señalar explícitamente las locaciones iniciales nos permitirá homogeneizar la noción de predicado sobre un sistema de transiciones.

Un *sistema de transiciones* TS, será una 4-upla que contiene un conjunto de locaciones, un conjunto finito de sentencias $\mathcal{S}$, una relación en $\mathsf{Tran}_{\mathcal{L},\mathcal{S}}$ y un conjunto de estados iniciales, esto es:

$$\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta) \ ,$$

con $\mathcal{S} \subseteq \mathsf{Sent}$, $\mathcal{T} : \mathsf{Tran}_{\mathcal{L},\mathcal{S}}$ y $\Theta : \mathsf{Pred}_{\mathcal{L}}$.

A continuación veremos de que forma se pueden modelar programas con sistemas de transiciones.

Ejemplo 1.15
Comenzaremos mostrando cómo modelar un programa simple. El siguiente inicializa la variable n y después comienza un bucle donde esta se incrementa hasta llegar al valor N no negativo:

Programa 1.1

$\{N \geqslant 0\}$
$n := 0;$
do $n < N \mapsto$
  $n := n + 1$
od

El primer paso de modelado consiste en definir las locaciones como puntos de ejecución en el programa. En este ejemplo alcanza con definir las locaciones l_0, l_1 y l_2, como se muestra en el siguiente programa.

Programa 1.2

l_0: $\{N \geqslant 0\}$
 $n := 0;$
l_1: **do** $n < N \mapsto$
  $n := n + 1$
 od
l_2:

Las locaciones l_0, l_1 y l_2 indican el comienzo del programa, la condición del bucle y la finalización del programa respectivamente. Con ello el conjunto de locaciones será definido como $\mathcal{L} = \{l_0, l_1, l_2\}$.

A continuación, para definir la relación $\mathcal{T}$ se procede a generar una transición por cada par de locaciones según el flujo del programa en cuestión. En nuestro caso basta con una transición de l_0 a l_1 ejecutando la inicialización del bucle, una transición de l_1 en si mismo ejecutando el cuerpo del bucle siempre y cuando se cumpla la condición, y una transición desde l_1 a l_2 en la finalización del bucle cuando esa condición no se cumpla:

$$\begin{aligned} \mathcal{T} \ \doteq \ \langle \lambda\, l_s, s, l_t \bullet \ & l_s = l_0 \wedge l_t = l_1 \wedge s = (n := 0) \\ \vee\ & l_s = l_1 \wedge l_t = l_1 \wedge s = (\square\, n < N \mapsto n := n + 1) \\ \vee\ & l_s = l_1 \wedge l_t = l_2 \wedge s = (\square\, n \geqslant N \mapsto \textbf{skip})\rangle \ . \end{aligned}$$

También $\mathcal{T}$ puede ser definido como un conjunto de transiciones:

$$\begin{aligned} \mathcal{T} \ \doteq \ \{ & (l_0,\, n := 0,\, l_1), \\ & (l_1,\, \square\, n < N \mapsto n := n + 1,\, l_1), \\ & (l_1,\, \square\, n \geqslant N \mapsto \textbf{skip},\, l_2) \} \ . \end{aligned}$$

Como el programa comienza en la locación l_0 , el conjunto de estados iniciales Θ será definido como el predicado $N \geqslant 0$ en la misma y false (o el conjunto vacío de estados) en las demás locaciones:

$$\begin{aligned} \Theta \ \doteq \ \langle \lambda\, l \bullet (\ & l = l_0 \mapsto N \geqslant 0 \\ \square\ & l \neq l_0 \mapsto \mathsf{false} \\ )\rangle & \ . \end{aligned}$$

o de igual manera

$$\Theta.l \ \doteq \ (l = l_0) \ \cap \ N \geqslant 0 \ .$$

Por lo tanto, el programa 1.1 puede ser modelado con el sistema de transiciones $\mathsf{TS} = (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ con $\mathcal{L}$, $\mathcal{T}$ y Θ definidos anteriormente y el conjunto de sentencias $\mathcal{S}$ formado por aquellas que aparecen en la segunda coordenada de cada transición:

$$\mathcal{S} = \{n := 0,\ \square\, n < N \mapsto n := n + 1,\ \square\, n \geqslant N \mapsto \textbf{skip}\} \ . \qquad \square$$

Ejemplo 1.16
A continuación desarrollaremos un ejemplo donde mostraremos la posibilidad de modelar un programa con saltos no estructurados. Sea el programa con las locaciones agregadas:

Programa 1.3

---

l_0: $\{N > 0\}$
  do $y < N \mapsto$
    $x := x - y$;
l_1: **if** $x \leqslant 0 \mapsto$ **goto** l_2
    $\Box\ x \geqslant 0 \mapsto y := y + 1$
    fi
  od
l_2:

Sobre este programa, el sistema de transiciones constará de las locaciones $\mathcal{L} = \{l_0, l_1, l_2\}$, con una transición de l_0 a l_1 (ingresando al bucle y ejecutando la asignación a la variable x), una transición de l_0 a l_2 (por la finalización del bucle), una transición de l_1 a l_2 (por la ejecución del salto no estructurado) y una transición de l_1 a l_0 (ejecutando la asignación a la variable y). Esto es

$$\begin{aligned} \mathcal{T} \doteq \{ & (l_0, \Box\, y < N \mapsto x := x - y,\, l_1), \\ & (l_0, \Box\, y \geqslant N \mapsto \mathbf{skip},\, l_2), \\ & (l_1, \Box\, x \leqslant 0 \mapsto \mathbf{skip},\, l_2), \\ & (l_1, \Box\, x \geqslant 0 \mapsto y := y + 1,\, l_0) \} \ . \end{aligned}$$

El conjunto de estados iniciales Θ será definido como el predicado $N > 0$ en la locación l_0 y false en las demás locaciones:

$$\Theta.l \doteq (l = l_0) \cap N > 0 \ .$$

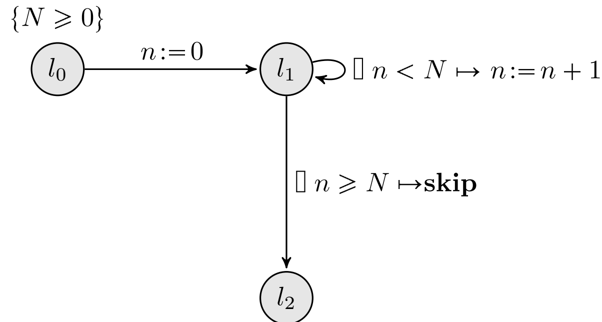
Por último, el conjunto $\mathcal{S}$ estará formado por las sentencias que aparecen como segundas coordenadas de cada transición:

$$\begin{aligned} \mathcal{S} \doteq \{ & \Box\, y < N \mapsto x := x - y,\ \Box\, y \geqslant N \mapsto \mathbf{skip}, \\ & \Box\, x \leqslant 0 \mapsto \mathbf{skip},\ \Box\, x \geqslant 0 \mapsto y := y + 1 \} \ . \end{aligned}$$ □

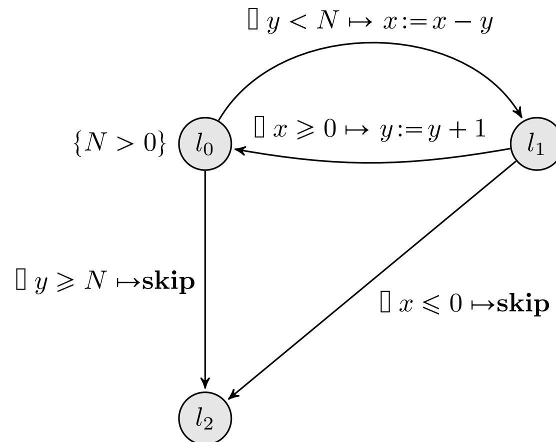
1.5.1 Gráfico de transiciones

Una manera usual de representar un sistema de transiciones es mediante un grafo. En el mismo, cada nodo representa una locación y sus aristas denotan las transiciones particulares: las aristas se etiquetan con la sentencia correspondiente a la transición representada, donde sus nodos de salida y llegada serán las locaciones de salida y llegada respectivamente. También se suele agregar el predicado $\Theta.l$ a los nodos que representan locaciones de entrada. Los grafos de transiciones 1.1 y 1.2 (pág. 21) representan los sistemas de transiciones de los programas 1.1 y 1.3 respectivamente.

Grafo de transiciones 1.1 Programa 1.1



Grafo de transiciones 1.2 Programa 1.3



1.5.2 Sistemas de transiciones ejecutables

Al modelar programas con sistemas de transiciones una pregunta que nos podemos hacer es si el conjunto total de programas modelados son ejecutables en una computadora. Una restricción básica que poseen estos programas es que no debe ser posible elegir un número infinito de alternativas en un tiempo finito. Si la máquina intenta hacer esta elección, la ejecución podría no terminar, ya que debe asignar una mínima cantidad de tiempo en cada caso, o de otra forma podría solo considerar un subconjunto finito de las posibles elecciones. Observando la definición del tipo Sent como relaciones arbitrarias, si la imagen de una relación posee cardinalidad infinita esta condición claramente no se cumple ya que desde un estado la ejecución necesitaría poder realizarse eligiendo entre una cantidad infinita de alternativas.

Esta ausencia en la ejecutabilidad de los programas está ligada a la falta de or-continuidad del transformador de weakest precondition (de manera dual and-continuidad de sp) como vimos en la sección 1.4.3. En [BAW98, capítulo 22] se describe este fenómeno, mostrando que la propiedad de continuidad en la

weakest precondition de relaciones se comprueba únicamente si tienen imagen finita (cada estado está relacionado solo con un conjunto finito de estados).

En nuestro trabajo podemos simplificar aún más esta condición sin limitar el poder expresivo de los sistemas de transiciones imponiendo como restricción que las sentencias en Sent sean deterministas: como las sentencias son los bloques constitutivos de los sistemas de transiciones el no determinismo puede modelarse con múltiples transiciones deterministas desde una misma locación de la forma que se hizo en los ejemplos. La finitud de las elecciones desde una locación está garantizada por el hecho de que en un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ la relación $\mathcal{T}$ es finita. Sumado al hecho de que cualquier sentencia determinista puede ser escrita como una suposición compuesta con una actualización funcional (ejemplo 1.1), para modelar programas podemos restringir el conjunto finito $\mathcal{S}$ a sentencias de la forma $[b];\langle f\rangle$, o como una asignación guardada $\square\ \mathsf{B} \mapsto v_1, \cdots, v_k := e_1, \cdots, e_k$ si la cantidad de variables es finita. De todas maneras, esta caracterización solo es posible cuando estamos modelando programas ejecutables. En este capítulo continuaremos trabajando con sentencias generales para simplificar los resultados. En próximos capítulos volveremos a señalar esta restricción.

1.6 Transformadores de predicados en sistemas de transiciones

En la sección 1.4 definimos transformadores de predicados sobre el dominio Pred. En esta sección generalizaremos esos operadores al dominio $\mathsf{Pred}_{\mathcal{L}}$ aprovechando que el mismo es la extensión puntual del anterior, y por lo tanto es un reticulado completo y booleano ya que hereda estas propiedades por ser la extensión puntual de Pred y además es atómico.

A partir de este hecho, extenderemos punto a punto las operaciones definidas sobre Pred (sección 1.2) a $\mathsf{Pred}_{\mathcal{L}}$. Dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$, sean P Q predicados en $\mathsf{Pred}_{\mathcal{L}}$ y $l \in \mathcal{L}$, definimos la siguientes operaciones:

$$
\begin{array}{rcll}
\mathsf{True}.l & \doteq & \mathsf{true} & (\top \text{ en } \mathsf{Pred}_{\mathcal{L}}) \\
\mathsf{False}.l & \doteq & \mathsf{false} & (\bot \text{ en } \mathsf{Pred}_{\mathcal{L}}) \\
(P \cap Q).l & \doteq & P.l \cap Q.l & (\sqcap \text{ en } \mathsf{Pred}_{\mathcal{L}}) \\
(P \cup Q).l & \doteq & P.l \cup Q.l & (\sqcup \text{ en } \mathsf{Pred}_{\mathcal{L}}) \\
(\neg P).l & \doteq & \neg P.l & (\neg \text{ en } \mathsf{Pred}_{\mathcal{L}}) \\
(P \equiv Q).l & \doteq & P.l \equiv Q.l & (\equiv \text{ en } \mathsf{Pred}_{\mathcal{L}}) \\
(P \Rightarrow Q).l & \doteq & P.l \Rightarrow Q.l & (\Rightarrow \text{ en } \mathsf{Pred}_{\mathcal{L}})
\end{array}
$$

$$
\begin{array}{rcll}
P \subseteq Q & \doteq & \langle \forall\, l \,:\, l \in \mathcal{L} :\ P.l \subseteq Q.l \rangle & (\text{orden en } \mathsf{Pred}_{\mathcal{L}}) \\
P = Q & \doteq & \langle \forall\, l \,:\, l \in \mathcal{L} :\ (P = Q).l \rangle & (\text{igualdad en } \mathsf{Pred}_{\mathcal{L}})
\end{array}
$$

Los ínfimos y supremos en $\mathsf{Pred}_{\mathcal{L}}$ se definen punto a punto de forma análoga. Sean $I \subseteq \Gamma$ conjuntos cualesquiera y $P : \Gamma \to \mathsf{Pred}_{\mathcal{L}}$, entonces:

$$
\begin{array}{rcll}
\langle \bigcap\, i \,:\, I.i :\ P.i \rangle .l & \doteq & \langle \bigcap\, i \,:\, I.i :\ P.i.l \rangle & (\text{ínfimo en } \mathsf{Pred}_{\mathcal{L}}) \\
\langle \bigcup\, i \,:\, I.i :\ P.i \rangle .l & \doteq & \langle \bigcup\, i \,:\, I.i :\ P.i.l \rangle & (\text{supremo en } \mathsf{Pred}_{\mathcal{L}})
\end{array}
$$

De aquí en adelante, en el caso que el conjunto $\mathcal{L}$ esté numerado (como en los ejemplos anteriores) denotaremos un predicado el $\mathsf{Pred}_{\mathcal{L}}$ como una lista de predicados (en Pred) donde cada posición contendrá su valor en cada locación. Así, en el ejemplo 1.15 el conjunto de estados iniciales será: $\Theta \doteq [N \geqslant 0, \mathsf{false}, \mathsf{false}]$. Claramente las operaciones en $\mathsf{Pred}_{\mathcal{L}}$ pueden ser instrumentadas aplicando punto a punto las definidas en Pred sobre cada elemento de estas listas.

1.6.1 Strongest postcondition

Dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$, el transformador strongest postcondition sobre ese sistemas de transiciones será un operador que toma el conjunto $\mathcal{T}$ de transiciones, un predicado en el sistema y devuelve un predicado en el mismo:

$$\begin{array}{|l} \mathrm{SP} : \mathsf{Tran}_{\mathcal{L},\mathcal{S}} \to \mathsf{Pred}_{\mathcal{L}} \to \mathsf{Pred}_{\mathcal{L}} \\ \hline \mathrm{SP}.\mathcal{T}.P.l' \doteq \langle \bigcup l, s \ : \mathcal{T}.l.s.l' : \ \mathrm{sp}.s.(P.l) \rangle \end{array}$$

Para cada locación, este transformador devuelve la unión (supremo) de la transformación sp sobre las transiciones entrantes a la misma. De forma intuitiva el resultado del transformador se puede caracterizar como las configuraciones (estados/locaciones) alcanzables mediante un paso de ejecución del programa desde las configuraciones del predicado parámetro.

Ejemplo 1.17
Sea el sistema de transiciones correspondiente al programa 1.1 (grafo de transiciones 1.1). Calcularemos el resultado de este transformador con respecto al estado inicial $\Theta \doteq [N \geqslant 0, \mathsf{false}, \mathsf{false}]$. Para ello deberemos obtener el supremo de aplicar sp sobre cada transición entrante a cada locación.

Comencemos por calcular $\mathrm{SP}.\mathcal{T}.\Theta.l_0$. En este caso no existen transiciones cuya locación de llegada sea l_0; esto es $\neg\mathcal{T}.l.s.l_0$ para todo l y s. Por lo tanto el transformador en l_0 es el supremo con rango vacío:

$$\mathrm{SP}.\mathcal{T}.\Theta.l_0 = \mathsf{false}$$

La locación l_1 tiene dos transiciones entrantes: una parte de l_0 y otra de l_1. En estas locaciones Θ vale $N \geqslant 0$ y false respectivamente. Para esta última el transformador sp dentro del supremo devuelve false ya que el mismo es estricto. Calculemos el resultado sobre la primera locación:

$$\begin{array}{ll}
& \mathrm{sp}.(n{:=}0).(N \geqslant 0).\sigma' \\
\equiv & \{ \text{ Desplegado de abreviaciones } \} \\
& \mathrm{sp}.\langle\langle \lambda\sigma \bullet \sigma[n \mapsto 0] \rangle\rangle.\langle \lambda\sigma \bullet \sigma.N \geqslant 0 \rangle.\sigma' \\
\equiv & \{ \text{ Definición de sp } \} \\
& \langle \exists\, \sigma \ : \sigma.N \geqslant 0 : \ \sigma[n \mapsto 0] = \sigma' \rangle \\
\equiv & \{ \text{ Actualización de } \sigma \text{ y separación de término } \} \\
& \langle \exists\, \sigma \ : \sigma.N \geqslant 0 \ : \\
& \qquad \sigma'.n = 0 \wedge \sigma'.N = \sigma.N \wedge \langle \forall\, v \in \mathsf{Var} \ : v \neq n \wedge v \neq N : \ \sigma.v = \sigma'.v \rangle \rangle \\
\equiv & \{ \text{ Intercambio, lógica, distributividad } \} \\
& \sigma'.N \geqslant 0 \wedge \sigma'.n = 0 \wedge \\
& \langle \exists\, \sigma \ :: \ \sigma'.N = \sigma.N \wedge \langle \forall\, v \in \mathsf{Var} \ : v \neq n \wedge v \neq N : \ \sigma.v = \sigma'.v \rangle \rangle
\end{array}$$

$\equiv$ { Instanciación σ por σ' }

$\quad \sigma'.N \geqslant 0 \wedge \sigma'.n = 0$.

Por lo tanto

$$\text{sp}.(n:=0).(N \geqslant 0) = N \geqslant 0 \wedge n = 0$$

y este resultado será el mismo que el del transformador SP en la locación l_1 :

$$\text{SP}.\mathcal{T}.\Theta.l_1 = N \geqslant 0 \wedge n = 0 \ .$$

La locación l_2 tiene solo una transición entrante desde l_1. Pero como Θ es falso en l_1 y sp es estricto obtenemos:

$$\text{SP}.\mathcal{T}.\Theta.l_2 = \mathsf{false} \ .$$

El resultado final de aplicar el transformador será entonces

$$\text{SP}.\mathcal{T}.\Theta = [\mathsf{false}, N \geqslant 0 \wedge n = 0, \mathsf{false}]$$

el cual muestra que los estados alcanzables desde los estados iniciales en Θ después de un paso de ejecución del programa están únicamente en la locación l_1, antes de comenzar el bucle. De esta forma el programa con las anotaciones parciales obtenidas será:

Programa 1.4

l_0: $\{N \geqslant 0\}$
  $n := 0$;
l_1: $\{N \geqslant 0 \wedge n = 0\}$
  do $n < N \mapsto$
    $n := n + 1$
  od
l_2:

□

1.6.2 Weakest liberal precondition

El transformador *weakest liberal precondition* puede ser extendido también a predicados sobre sistemas de transiciones:

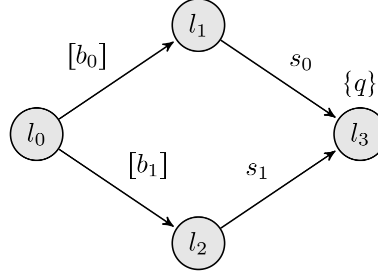
$$\text{WLP} : \mathsf{Tran}_{\mathcal{L},\mathcal{S}} \to \mathsf{Pred}_{\mathcal{L}} \to \mathsf{Pred}_{\mathcal{L}}$$
$$\text{WLP}.\mathcal{T}.Q.l \ \doteq \ \langle \bigcap l', s \ : \mathcal{T}.l.s.l' : \ \text{wlp}.s.(Q.l') \rangle$$

El mismo en cada locación devuelve la intersección (ínfimo) de la aplicación del transformador wlp sobre cada transición saliente desde la misma. El resultado de este transformador puede caracterizarse de forma intuitiva como el conjunto de configuraciones (estados/locaciones) tales que si en el sistema de transiciones se ejecuta una sentencia desde alguna de ellas, si la sentencia termina lo hace a alguna configuración perteneciente a Q.

Ejemplo 1.18
Consideremos el programa 1.5 junto con su grafo de transiciones. En el mismo se ha anotado el final del programa (locación l_3) con la poscondición q.

Programa 1.5

l_0:
  if $b_0 \mapsto$
l_1: s_0
  $\square$ $b_1 \mapsto$
l_2: s_1
  fi
l_3: $\{q\}$



En este ejemplo obtendremos el conjunto de estados a partir de los cuales si el programa termina en l_3 lo hace en un estado que satisface q. Notar que no interesa si el programa alcanza otra locación distinta a l_3. Si alcanza estas locaciones puede hacerlo en cualquier estado pero si llega a l_3 debe cumplir q. Este resultado puede ser obtenido aplicando el transformador WLP al predicado $Q \doteq [\mathsf{true}, \mathsf{true}, \mathsf{true}, q]$.

Sea $\mathcal{T}$ el sistema de transiciones descripto por el grafo del programa 1.5. Calcularemos el resultado del transformador en cada locación. Desde l_0 solo hay dos transiciones salientes hacia l_1 y l_2 por lo que el cálculo del ínfimo de la definición de WLP se reduce al cálculo de una intersección:

$$
\begin{aligned}
& \mathrm{WLP}.\mathcal{T}.Q.l_0 \\
= \; & \{ \text{ Definición de } Q \text{ } \} \\
& \mathrm{WLP}.\mathcal{T}.[\mathsf{true}, \mathsf{true}, \mathsf{true}, q].l_0 \\
= \; & \{ \text{ Definición de WLP } \} \\
& \mathrm{wlp}.[b_0].\mathsf{true} \;\cap\; \mathrm{wlp}.[b_1].\mathsf{true} \\
= \; & \{ \text{ wlp es no interrumpible } \} \\
& \mathsf{true}
\end{aligned}
$$

Desde l_1 (respectivamente l_2) solo hay una transición saliente hacia l_3, por lo que el resultado es simplemente $\mathrm{wlp}.s_0.q$ (respectivamente $\mathrm{wlp}.s_1.q$). Desde l_3 no hay transiciones salientes por lo que el ínfimo es true ($\top$ del reticulado Pred). El resultado final será:

$$\mathrm{WLP}.\mathcal{T}.[\mathsf{true}, \mathsf{true}, \mathsf{true}, q] = [\mathsf{true}, \mathrm{wlp}.s_0.q, \mathrm{wlp}.s_1.q, \mathsf{true}] \;,$$

lo cual muestra que si después de un paso de ejecución el programa llega a l_3 en un estado de q, entonces si partió de l_1 lo hizo desde un estado en $\mathrm{wlp}.s_0.q$, o si partió de l_2 lo hizo desde un estado en $\mathrm{wlp}.s_1.q$.

El mismo razonamiento puede aplicarse para obtener que sucede después de dos pasos de ejecución. Calculando el resultado del transformador sobre el último resultado y teniendo en cuenta que $\mathrm{wlp}.[b].q = b \Rightarrow q$ obtenemos:

$$
\begin{aligned}
& \mathrm{WLP}.\mathcal{T}.[\mathsf{true}, \mathrm{wlp}.s_0.q, \mathrm{wlp}.s_1.q, \mathsf{true}] = \\
& \quad [b_0 \Rightarrow \mathrm{wlp}.s_0.q \;\cap\; b_1 \Rightarrow \mathrm{wlp}.s_1.q, \mathsf{true}, \mathsf{true}, \mathsf{true}]
\end{aligned}
$$

o, junto con el cálculo anterior:

$$(\mathrm{WLP}.\mathcal{T})^2.Q = [b_0 \Rightarrow \mathrm{wlp}.s_0.q \,\cap\, b_1 \Rightarrow \mathrm{wlp}.s_1.q, \mathsf{true}, \mathsf{true}, \mathsf{true}]$$

La interpretación de este resultado es la siguiente: si después de dos pasos de ejecución el programa llega a l_3 en un estado de q, entonces si partió de l_0 lo hizo desde un estado en $b_0 \Rightarrow \mathrm{wlp}.s_0.q \,\cap\, b_1 \Rightarrow \mathrm{wlp}.s_1.q$. Notar que este es el resultado clásico de la aplicación del transformador weakest liberal precondition a la sentencia condicional [DS90].

Utilizando los cálculos de WLP.$\mathcal{T}$ y $(\mathrm{WLP}.\mathcal{T})^2$ podemos realizar las anotaciones en el siguiente programa:

Programa 1.6

l_0: $\{b_0 \Rightarrow \mathrm{wlp}.s_0.q \,\cap\, b_1 \Rightarrow \mathrm{wlp}.s_1.q\}$
if $b_0 \mapsto$
l_1: $\{\mathrm{wlp}.s_0.q\}$
s_0
$\square$ $b_1 \mapsto$
l_2: $\{\mathrm{wlp}.s_1.q\}$
s_1
fi
l_3: $\{q\}$

□

1.6.3 Propiedades de los transformadores

En la sección 1.4.3 se definieron ciertas propiedades de los transformadores de predicados en general (monotonía, juntividad y continuidad) y vimos que los transformadores de predicados wlp y sp cumplen algunas de ellas. Estas propiedades fueron definidas en el reticulado Pred pero, como veremos en el próximo capítulo, pueden ser planteadas sobre otros reticulados y posets. Ya que $\mathsf{Pred}_{\mathcal{L}}$ es también un reticulado y los transformadores WLP y SP son funciones sobre el mismo, veremos como estas propiedades se extienden naturalmente a este dominio.

Propiedades 1.19 (Transformadores SP y WLP)
Dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$

- SP.$\mathcal{T}$ es universalmente disyuntivo:

$$\mathrm{SP}.\mathcal{T}.\langle \bigcup\, i \,:I.i:\, P.i \,\rangle = \langle \bigcup\, i \,:I.i:\, \mathrm{SP}.\mathcal{T}.(P.i) \,\rangle$$

para todo conjunto de índices $I \subseteq \Gamma$ y para todo $P : \Gamma \to \mathsf{Pred}_{\mathcal{L}}$. Por lo tanto es estricto ($\mathrm{SP}.\mathcal{T}.\mathsf{False} = \mathsf{False}$).

Esta propiedad indica que el transformador SP es distributivo con respecto a uniones arbitrarias de elementos en $\mathsf{Pred}_{\mathcal{L}}$. Con esta propiedad y la jerarquía dada en el teorema 1.11 se deduce que SP es también or-continuo y monótono.

- WLP.$\mathcal{T}$ es universalmente conjuntivo:

  $$\text{WLP}.\mathcal{T}.\langle \bigcap i\ : I.i :\ P.i \rangle = \langle \bigcap i\ : I.i :\ \text{WLP}.\mathcal{T}.(P.i) \rangle$$

  para todo conjunto de índices $I \subseteq \Gamma$ y para todo $P : \Gamma \to \mathsf{Pred}_{\mathcal{L}}$. Por lo tanto es no interrumpible (WLP.$\mathcal{T}$.True = True).

  Esta propiedad indica que el transformador WLP es distributivo con respecto a intersecciones arbitrarias de elementos en $\mathsf{Pred}_{\mathcal{L}}$. Con esta propiedad y la jerarquía dada en el teorema 1.11 se deduce que WLP es también and-continuo y monótono. □

En la sección 1.4.3 (propiedades 1.12) vimos que los transformadores sp y wlp eran positivamente disyuntivos y conjuntivos respectivamente solo si la sentencia era determinista. Para extender esta propiedades a SP y WLP primero necesitamos definir este concepto en el marco de los sistemas de transiciones.

Definición 1.20 (Determinismo)
Un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ será determinista cuando todas sus sentencias $s \in \mathcal{S}$ sean deterministas ($\mathsf{Card}.s \leqslant 1$) y en toda locación las sentencias en cada par de sus transiciones salientes tengan dominios disjuntos. Esto es, para todo par de transiciones distintas y con igual locación de salida (l, s, l') y $(l, \tilde{s}, \tilde{l})$ se cumple $\mathsf{Dom}.s \cap \mathsf{Dom}.\tilde{s} = \varnothing$.

Propiedades 1.21 (SP y WLP deterministas)
Dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$

- Si TS es determinista, el transformador SP.$\mathcal{T}$ es positivamente conjuntivo.
- Si TS es determinista, WLP.$\mathcal{T}$ es positivamente disyuntivo. □

Todas estas propiedades pueden ser demostradas utilizando las propiedades 1.12 y la definición de los transformadores.

A partir del ejemplo 1.18 puede vislumbrarse que las 3-uplas de Hoare pueden extenderse a sistemas de transiciones. Por ejemplo la proposición $\{\Theta\}\ \mathcal{T}\ \{Q\}$ puede interpretarse como $\Theta \subseteq \text{WLP}.\mathcal{T}.Q$. Al igual que con sp es posible utilizar el transformador SP para denotar esta misma proposición:

Propiedad 1.22 (SP WLP equivalencia)
Para todo sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ y predicados P, Q se cumple:

$$P \subseteq \text{WLP}.\mathcal{T}.Q \ \equiv\ \text{SP}.\mathcal{T}.P \subseteq Q\ .$$ □

De aquí en adelante usaremos la notación $\{P\}\ \mathcal{T}\ \{Q\}$ para indicar las condiciones booleanas $P \subseteq \text{WLP}.\mathcal{T}.Q$ o $\text{SP}.\mathcal{T}.P \subseteq Q$ indistintamente.

La dualidad entre los transformadores sp y wlp (propiedad 1.14) también tiene su correlato en el contexto actual:

Propiedad 1.23 (Dualidad SP y WLP)
Dado el sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$, definimos la inversa del conjunto de transiciones $\mathcal{T}$ como

$$\mathcal{T}^{-1}.l.s.l' = \mathcal{T}.l'.s^{-1}.l$$

para todo $l, l' \in \mathcal{L}$ y $s \in \mathcal{S}$ (de forma intuitivamente el grafo de $\mathcal{T}^{-1}$ tiene los mismos nodos que $\mathcal{T}$, con las flechas en dirección opuesta y con las sentencias inversas).

Entonces se cumple:

$$\text{SP}.\mathcal{T}.P = \neg(\text{WLP}.\mathcal{T}^{-1}.\neg P) \ . \qquad \square$$

La demostración es directa por la propiedad 1.14 y la definición de los transformadores.

1.7 Semántica operacional

En las secciones anteriores hemos hablado de pasos de ejecución de un sistema de transiciones de manera informal. En esta sección aclararemos este concepto especificando una semántica operacional de los sistemas de transiciones. De esta manera veremos cómo las ejecuciones describen el comportamiento operacional de los sistemas de transiciones a través de los pasos de ejecución del programa que representa. Para comenzar definiremos formalmente el concepto de configuración. Una configuración será un par donde la primera componente es un estado y la segunda una locación. La misma representará el valor de las variables junto con el valor del contador de programa (locación) en un momento de la ejecución.

Definición 1.24 (Configuración)
Llamaremos *conjunto de configuraciones* al conjunto $\Gamma \doteq \mathcal{L} \times \Sigma$ (pares locaciones y estados) y a cada elemento del mismo una *configuración*. $\square$

Además, representaremos la transformación de las configuraciones por cada paso de ejecución de un programa.

Definición 1.25 (Relación de transición)
Sea $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ un sistema de transiciones. Llamaremos *relación de transición* de TS a la definida como

$$\begin{array}{|l} \leadsto : \Gamma \to \Gamma \to \mathsf{Bool} \\ \hline (l,\sigma) \leadsto (l',\sigma') \doteq \langle\, \exists\, s \,:\, \mathcal{T}.l.s.l' \,:\, s.\sigma.\sigma' \,\rangle \end{array} \qquad \square$$

Por ultimo definiremos el concepto de ejecución en un sistema de transiciones.

Definición 1.26 (Ejecución)
Sea $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ un sistema de transiciones. Una *ejecución finita* de TS será una secuencia finita de configuraciones $\varrho : [0..n] \to (\mathcal{L} \times \Sigma)$ con $n \in \mathbb{N}$ que comienza en Θ y cumple la relación de transición, esto es:

$$\begin{array}{l} \Theta.l_0.\sigma_0 \wedge \langle\, \forall\, i \,:\, 0 \leqslant i < n \,:\, \varrho.i \leadsto \varrho.(i+1) \,\rangle \\ \quad [\![(\sigma_0, l_0) = \varrho.0]\!] \end{array}$$

Notar que bajo esta definición una configuración (l, σ) es por sí sola una ejecución si σ satisface $\Theta.l$.

Una *ejecución infinita* ϱ de TS será una secuencia infinita de configuraciones $\varrho : \mathbb{N} \to (\mathcal{L} \times \Sigma)$ que comienza en Θ y cumple la relación de transición, esto es:

$$\Theta.l_0.\sigma_0 \wedge \langle \forall\, i \ : i \geqslant 0 : \ \varrho.i \rightsquigarrow \varrho.(i+1) \rangle$$
$$[\![(\sigma_0, l_0) = \varrho.0]\!]$$

Llamaremos de forma genérica *ejecución* tanto a ejecuciones finitas como infinitas. □

Con esta caracterización definiremos la semántica operacional de un sistema de transiciones:

Definición 1.27 (Semántica operacional)
Dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ su semántica operacional estará dada por el conjunto de todas las ejecuciones posibles:

$$[\![\mathsf{TS}]\!] = \{\varrho \mid \varrho \text{ es ejecución de } \mathsf{TS}\} \ .$$ □

En la sección 1.6 se mencionó de manera informal el comportamiento de los transformadores SP y WLP. Ahora, dentro de este contexto operacional, estos transformadores pueden ser descriptos formalmente. Veamos primero el caso del transformador SP. Dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$, $P : \mathsf{Pred}_{\mathcal{L}}$, $l' : \mathcal{L}$ y $\sigma : \Sigma$

$$\begin{aligned}
& \mathrm{SP}.\mathcal{T}.P.l'.\sigma' \\
\equiv\ & \{ \text{ Definición de SP } \} \\
& \langle \textstyle\bigcup\, l, s \ : \mathcal{T}.l.s.l' : \ \mathrm{sp}.s.(P.l) \rangle .\sigma' \\
\equiv\ & \{ \text{ Definición de supremo } \} \\
& \langle \exists\, l, s \ : \mathcal{T}.l.s.l' : \ \mathrm{sp}.s.(P.l).\sigma' \rangle \\
\equiv\ & \{ \text{ Definición de sp } \} \\
& \langle \exists\, l, s \ : \mathcal{T}.l.s.l' : \ \langle \exists\, \sigma \ : P.l.\sigma : \ s.\sigma.\sigma' \rangle \rangle \\
\equiv\ & \{ \text{ Anidado } \} \\
& \langle \exists\, l, s, \sigma \ : \mathcal{T}.l.s.l' \wedge P.l.\sigma : \ s.\sigma.\sigma' \rangle \\
\equiv\ & \{ \text{ Anidado } \} \\
& \langle \exists\, l, \sigma \ : P.l.\sigma : \ \langle \exists\, s \ : \mathcal{T}.l.s.l' : \ s.\sigma.\sigma' \rangle \rangle \\
\equiv\ & \{ \text{ Definición de relación de transición } \} \\
& \langle \exists\, l, \sigma \ : P.l.\sigma : \ (l, \sigma) \rightsquigarrow (l', \sigma') \rangle \ .
\end{aligned}$$

Por lo tanto

$$\mathrm{SP}.\mathcal{T}.P.l'.\sigma' = \langle \exists\, l, \sigma \ : P.l.\sigma : \ (l, \sigma) \rightsquigarrow (l', \sigma') \rangle$$

lo cual muestra que el transformador SP devuelve las configuraciones alcanzables en un paso de ejecución desde el predicado P.

Realicemos el mismo análisis para el transformador WLP:

$$
\begin{aligned}
& \text{WLP}.\mathcal{T}.Q.l.\sigma \\
\equiv\ & \{ \text{ Definición de WLP } \} \\
& \langle \textstyle\bigcap l', s\ : \mathcal{T}.l.s.l' :\ \text{wlp}.s.(Q.l') \rangle .\sigma \\
\equiv\ & \{ \text{ Definición de ínfimo } \} \\
& \langle \forall\, l', s\ : \mathcal{T}.l.s.l' :\ \text{wlp}.s.(Q.l').\sigma \rangle \\
\equiv\ & \{ \text{ Definición de wlp } \} \\
& \langle \forall\, l', s\ : \mathcal{T}.l.s.l' :\ \langle \forall\, \sigma'\ : s.\sigma.\sigma' :\ Q.l'.\sigma' \rangle \rangle \\
\equiv\ & \{ \text{ Anidado } \} \\
& \langle \forall\, l', s, \sigma'\ : \mathcal{T}.l.s.l' \wedge s.\sigma.\sigma' :\ Q.l'.\sigma' \rangle \\
\equiv\ & \{ \text{ Partición de rango generalizada } \} \\
& \langle \forall\, l', \sigma'\ : \langle \exists\, s\ : \mathcal{T}.l.s.l' :\ s.\sigma.\sigma' \rangle :\ Q.l'.\sigma' \rangle \\
\equiv\ & \{ \text{ Definición de relación de transición } \} \\
& \langle \forall\, l', \sigma'\ : (l,\sigma) \rightsquigarrow (l',\sigma') :\ Q.l'.\sigma' \rangle \\
\equiv\ & \{ \text{ Intercambio } \} \\
& \langle \forall\, l', \sigma'\ ::\ (l,\sigma) \rightsquigarrow (l',\sigma')\ \Rightarrow\ Q.l'.\sigma' \rangle \ .
\end{aligned}
$$

Por lo tanto

$$\text{WLP}.\mathcal{T}.Q.l.\sigma \equiv \Big\langle \forall\, l', \sigma'\ ::\ (l,\sigma) \rightsquigarrow (l',\sigma')\ \Rightarrow\ Q.l'.\sigma' \Big\rangle$$

lo cual muestra que si se realiza un paso de ejecución desde una configuración que pertenece al resultado del transformador ($\text{WLP}.\mathcal{T}.Q$), se termina en el predicado Q. Esta descripción de SP y WLP por medio de la semántica operacional coincide con la dada informalmente en la sección 1.6. El próximo teorema plasma el resultado de estas dos demostraciones:

Teorema 1.28 (Transformadores en función de transiciones)
Dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$, los transformadores strongest postcondition y weakest precondition pueden ser descriptos como:

$$
\begin{aligned}
\text{WLP}.\mathcal{T}.Q.l.\sigma &\equiv \langle \forall\, l', \sigma'\ : (l,\sigma) \rightsquigarrow (l',\sigma') :\ Q.l'.\sigma' \rangle \ \ y \\
\text{SP}.\mathcal{T}.P.l'.\sigma' &\equiv \langle \exists\, l, \sigma\ : P.l.\sigma :\ (l,\sigma) \rightsquigarrow (l',\sigma') \rangle \ .
\end{aligned}
$$

□

Para reforzar aún más esta relación entre la semántica operacional y la descripta en las secciones anteriores, mostraremos que el conjunto de configuraciones resultante de aplicar n veces el transformador strongest postcondition sobre las configuraciones iniciales Θ, está formado por las n-ésimas configuraciones de las ejecuciones. El siguiente teorema describe esta propiedad:

Teorema 1.29
Sea $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ un sistema de transiciones. Entonces para toda configuración (l', σ') se verifica

$$(\text{SP}.\mathcal{T})^n.\Theta.l'.\sigma' \equiv \Big\langle \exists\, \varrho\ : \varrho \in [\![\mathsf{TS}]\!] \wedge n \in \mathsf{Dom}.\varrho :\ \varrho.n = (l',\sigma') \Big\rangle$$

con $\mathsf{Dom}.\varrho$ el dominio de ϱ. □

Demostración

Demostraremos este teorema por inducción en n. En la demostración utilizaremos la notación de listas para denotar secuencias al estilo [BSB08]. En particular usaremos el operador $\varrho\uparrow n$ que devuelve la lista de los primeros n elementos de una secuencia y el operador $\varrho \lhd (l,\sigma)$ que pega un elemento al final de una secuencia finita.

Caso base ($n = 0$)

Demostraremos la doble implicación.

($\Rightarrow$)

$$\begin{array}{ll}
& \langle \exists\, \varrho \ : \varrho \in [\![\mathsf{TS}]\!] \wedge 0 \in \mathsf{Dom}.\varrho : \ \varrho.0 = (l',\sigma') \rangle \\
\Leftarrow & \{ \text{ Instanciación } \varrho = [(l',\sigma')] \ \} \\
& [(l',\sigma')] \in [\![\mathsf{TS}]\!] \wedge 0 \in \{0\} \wedge (l',\sigma') = (l',\sigma') \\
\equiv & \{ \text{ Lógica de predicados } \} \\
& [(l',\sigma')] \in [\![\mathsf{TS}]\!] \\
\Leftarrow & \{ \text{ Definición 1.26 de ejecución } \} \\
& \Theta.l'.\sigma'
\end{array}$$

($\Leftarrow$)

$$\begin{array}{ll}
& \langle \exists\, \varrho \ : \varrho \in [\![\mathsf{TS}]\!] \wedge 0 \in \mathsf{Dom}.\varrho : \ \varrho.0 = (l',\sigma') \rangle \Rightarrow \Theta.l'.\sigma' \\
\equiv & \{ \text{ Metateorema del testigo 1.44 (pág. 38) } \} \\
& \varrho \in [\![\mathsf{TS}]\!] \wedge 0 \in \mathsf{Dom}.\varrho \wedge \varrho.0 = (l',\sigma') \Rightarrow \Theta.l'.\sigma' \\
\equiv & \{ \text{ Definición 1.26 de ejecución } \} \\
& \mathsf{T}
\end{array}$$

Paso inductivo

Supongamos la hipótesis inductiva:

$$(\mathrm{SP}.\mathcal{T})^n.\Theta.l'.\sigma' \equiv \langle \exists\, \varrho \ : \varrho \in [\![\mathsf{TS}]\!] \wedge n \in \mathsf{Dom}.\varrho : \ \varrho.n = (l',\sigma') \rangle \ .$$

$$\begin{array}{ll}
& (\mathrm{SP}.\mathcal{T})^{n+1}.\Theta.l'.\sigma' \\
\equiv & \{ \text{ Desplegado de composición } \} \\
& \mathrm{SP}.\mathcal{T}.((\mathrm{SP}.\mathcal{T})^n.\Theta).l'.\sigma' \\
\equiv & \{ \text{ SP en función de transiciones (teorema 1.28) } \} \\
& \langle \exists\, l,\sigma \ : (\mathrm{SP}.\mathcal{T})^n.\Theta.l.\sigma : \ (l,\sigma) \rightsquigarrow (l',\sigma') \rangle \\
\equiv & \{ \text{ Hipótesis inductiva } \} \\
& \langle \exists\, l,\sigma \ : \langle \exists\, \varrho \ : \varrho \in [\![\mathsf{TS}]\!] \wedge n \in \mathsf{Dom}.\varrho : \ \varrho.n = (l,\sigma) \rangle : \ (l,\sigma) \rightsquigarrow (l',\sigma') \rangle \\
\equiv & \{ \text{ Partición de rango generalizada } \} \\
& \langle \exists\, l,\sigma,\varrho \ : \varrho \in [\![\mathsf{TS}]\!] \wedge n \in \mathsf{Dom}.\varrho \wedge \varrho.n = (l,\sigma) : \ (l,\sigma) \rightsquigarrow (l',\sigma') \rangle \\
\equiv & \{ \text{ Rango unitario } ((l,\sigma) = \varrho.n) \ \} \\
& \langle \exists\, \varrho \ : \varrho \in [\![\mathsf{TS}]\!] \wedge n \in \mathsf{Dom}.\varrho : \ \varrho.n \rightsquigarrow (l',\sigma') \rangle
\end{array}$$

Por lo tanto,

$$(\text{SP}.\mathcal{T})^{n+1}.\Theta.l'.\sigma' \equiv \langle \exists \varrho \ : \varrho \in [\![\mathsf{TS}]\!] \wedge n \in \mathsf{Dom}.\varrho : \ \varrho.n \rightsquigarrow (l',\sigma') \rangle$$

Para finalizar, demostraremos

$$\begin{array}{l} \langle \exists \varrho \ : \varrho \in [\![\mathsf{TS}]\!] \wedge n \in \mathsf{Dom}.\varrho : \ \varrho.n \rightsquigarrow (l',\sigma') \rangle \\ \equiv \\ \langle \exists \varrho \ : \varrho \in [\![\mathsf{TS}]\!] \wedge n+1 \in \mathsf{Dom}.\varrho : \ \varrho.(n+1) = (l',\sigma') \rangle \end{array}$$

por doble implicación y aplicando el metateorema del testigo [BSB08, metateorema 5.29] sobre los existenciales:

($\Rightarrow$)

$$\begin{array}{ll} & \varrho \in [\![\mathsf{TS}]\!] \wedge n \in \mathsf{Dom}.\varrho \wedge \varrho.n \rightsquigarrow (l',\sigma') \\ \Rightarrow & \{ \text{ Definición 1.26 de ejecución } \} \\ & \varrho{\uparrow}(n+1) \lhd (l',\sigma') \in [\![\mathsf{TS}]\!] \wedge n \in \mathsf{Dom}.\varrho \\ \equiv & \{ \text{ Manejo de secuencias } \} \\ & \varrho{\uparrow}(n+1) \lhd (l',\sigma') \in [\![\mathsf{TS}]\!] \wedge n+1 \in \mathsf{Dom}.(\varrho{\uparrow}(n+1) \lhd (l',\sigma')) \\ & \wedge (\varrho{\uparrow}(n+1) \lhd (l',\sigma')).(n+1) = (l',\sigma') \\ \Rightarrow & \{ \text{ Instanciación existencial de } \varrho \text{ por } \varrho{\uparrow}(n+1) \lhd (l',\sigma') \} \\ & \langle \exists \varrho \ :: \ \varrho \in [\![\mathsf{TS}]\!] \wedge n+1 \in \mathsf{Dom}.\varrho \wedge \varrho.(n+1) = (l',\sigma') \rangle \\ \Rightarrow & \{ \text{ Intercambio } \} \\ & \langle \exists \varrho \ : \varrho \in [\![\mathsf{TS}]\!] \wedge n+1 \in \mathsf{Dom}.\varrho : \ \varrho.(n+1) = (l',\sigma') \rangle \end{array}$$

Por lo tanto se demuestra

$$\begin{array}{l} \varrho \in [\![\mathsf{TS}]\!] \wedge n \in \mathsf{Dom}.\varrho \wedge \varrho.n \rightsquigarrow (l',\sigma') \\ \Rightarrow \\ \langle \exists \varrho \ : \varrho \in [\![\mathsf{TS}]\!] \wedge n+1 \in \mathsf{Dom}.\varrho : \ \varrho.(n+1) = (l',\sigma') \rangle \end{array}$$

y aplicando el metateorema del testigo tenemos

$$\begin{array}{l} \langle \exists \varrho \ : \varrho \in [\![\mathsf{TS}]\!] \wedge n \in \mathsf{Dom}.\varrho : \ \varrho.n \rightsquigarrow (l',\sigma') \rangle \\ \Rightarrow \\ \langle \exists \varrho \ : \varrho \in [\![\mathsf{TS}]\!] \wedge n+1 \in \mathsf{Dom}.\varrho : \ \varrho.(n+1) = (l',\sigma') \rangle \ . \end{array}$$

($\Leftarrow$)

$$\begin{array}{ll} & \varrho \in [\![\mathsf{TS}]\!] \wedge n+1 \in \mathsf{Dom}.\varrho \wedge \varrho.(n+1) = (l',\sigma') \\ \Rightarrow & \{ \text{ Definición 1.26 de ejecución } \} \\ & \varrho \in [\![\mathsf{TS}]\!] \wedge n \in \mathsf{Dom}.\varrho \wedge \varrho.n \rightsquigarrow (l',\sigma') \\ \Rightarrow & \{ \text{ Instanciación } \} \\ & \langle \exists \varrho \ : \varrho \in [\![\mathsf{TS}]\!] \wedge n \in \mathsf{Dom}.\varrho : \ \varrho.n \rightsquigarrow (l',\sigma') \rangle \end{array}$$

Por lo tanto se demuestra

$$\begin{array}{l} \varrho \in [\![\mathsf{TS}]\!] \wedge n+1 \in \mathsf{Dom}.\varrho \wedge \varrho.(n+1) = (l', \sigma') \\ \Rightarrow \\ \langle \exists\, \varrho \;:\; \varrho \in [\![\mathsf{TS}]\!] \wedge n \in \mathsf{Dom}.\varrho :\; \varrho.n \leadsto (l', \sigma') \rangle \end{array}$$

y aplicando el metateorema del testigo tenemos

$$\begin{array}{l} \langle \exists\, \varrho \;:\; \varrho \in [\![\mathsf{TS}]\!] \wedge n+1 \in \mathsf{Dom}.\varrho :\; \varrho.(n+1) = (l', \sigma') \rangle \\ \Rightarrow \\ \langle \exists\, \varrho \;:\; \varrho \in [\![\mathsf{TS}]\!] \wedge n \in \mathsf{Dom}.\varrho :\; \varrho.n \leadsto (l', \sigma') \rangle \;\;, \end{array}$$

con lo cual finaliza la prueba del teorema. □

Apéndice 1.A Propiedades de expresiones cuantificadas

En este apéndice daremos las propiedades de las expresiones cuantificadas que se utilizan en las demostraciones de este trabajo. Las mismas serán expuestas en la forma de axiomas, teoremas y metateoremas como en [BSB08].

Axioma 1.30 (Rango vacío)
Cuando el rango de especificación es vacío, la expresión cuantificada es igual al elemento neutro e del operador $\oplus$:

$$\langle \oplus i \ : \mathsf{F} : \ T \rangle = e$$

Si $\oplus$ no posee elemento neutro, la expresión no está bien definida.

Axioma 1.31 (Rango unitario)
Si el rango de especificación consiste en un solo elemento, la expresión cuantificada es igual al término evaluado en dicho elemento:

$$\langle \oplus i \ : i = N : \ T \rangle = T(i := N)$$

Otra forma de escribir esta regla es hacer explícita la dependencia de T de la variable i:

$$\langle \oplus i \ : i = N : \ T.i \rangle = T.N$$

Axioma 1.32 (Partición de rango)
Cuando el rango de especificación es de la forma $R \vee S$ y además se cumple al menos una de las siguientes condiciones:

– el operador $\oplus$ es idempotente,

– los términos booleanos R y S son disjuntos,

la expresión cuantificada puede reescribirse como sigue:

$$\langle \oplus i \ : R \vee S : \ T \rangle = \langle \oplus i \ : R : \ T \rangle \oplus \langle \oplus i \ : S : \ T \rangle$$

El hecho que la igualdad sea una relación simétrica, permite indistintamente reemplazar cualquiera de los dos miembros por el otro, vale decir que la regla de partición de rango puede leerse también de derecha a izquierda. Lo mismo ocurre con todas las reglas que siguen.

Axioma 1.33 (Partición de rango generalizada)
Si el operador $\oplus$ es idempotente y el rango de especificación es una cuantificación existencial, entonces:

$$\langle \oplus i \ : \langle \exists j \ : S.i.j : \ R.i.j \rangle : \ T.i \rangle = \langle \oplus i, j \ : S.i.j \wedge R.i.j : \ T.i \rangle$$

Axioma 1.34 (Regla del término)
Cuando el operador $\oplus$ aparece en el término de la cuantificación, la expresión cuantificada puede reescribirse de la siguiente manera:

$$\langle \oplus i \ : R : \ T \oplus G \rangle = \langle \oplus i \ : R : \ T \rangle \oplus \langle \oplus i \ : R : \ G \rangle$$

Axioma 1.35 (Regla del término constante)
Si el término de la cuantificación es constantemente igual a C, la variable cuantificada i no aparece en C, el operador $\oplus$ es idempotente y el rango de especificación es no vacío, entonces:

$$\langle \oplus i \ : R : \ C \rangle = C$$

Axioma 1.36 (Distributividad)
Si $\otimes$ es distributivo a izquierda con respecto a $\oplus$ y se cumple al menos una de las siguientes condiciones:

– el rango de especificación es no vacío,

– el elemento neutro del operador $\oplus$ existe y es absorbente para $\otimes$,

entonces:

$$\langle \oplus i \ : R : \ x \otimes T \rangle = x \otimes \langle \oplus i \ : R : \ T \rangle$$

Análogamente, si $\otimes$ es distributivo a derecha con respecto a $\oplus$ y se cumple al menos una de las condiciones anteriores, entonces:

$$\langle \oplus i \ : R : \ T \otimes x \rangle = \langle \oplus i \ : R : \ T \rangle \otimes x$$

Axioma 1.37 (Anidado)
Cuando hay más de una variable cuantificada y el rango de especificación es una conjunción de términos booleanos, uno de los cuales es independiente de alguna de las variables de cuantificación, la expresión cuantificada puede reescribirse de la siguiente manera:

$$\langle \oplus i, j \ : R.i \wedge S.i.j : \ T.i.j \rangle = \langle \oplus i \ : R.i : \ \langle \oplus j \ : S.i.j : \ T.i.j \rangle \rangle$$

Axioma 1.38 (Cambio de variable)
Si las variables en una secuencia j no se encuentran como variables en libres en el rango R ni el término T entonces pueden renombrarse las variables:

$$\langle \oplus i \ : R : \ T \rangle = \langle \oplus j \ : R(i := j) : \ T(i := j) \rangle$$

Este axioma puede también escribirse usando una referencia explícita a i:

$$\langle \oplus i \ : R.i : \ T.i \rangle = \langle \oplus j \ : R.j : \ T.j \rangle$$

Todas estas reglas pueden particularizarse, refiriéndose a operadores concretos. Es lo que haremos en adelante, con los operadores más usuales.

Teorema 1.39 (Cambio de variable)
Si f es una función que tiene inversa (es biyectiva) en el rango de especificación y j es una variable que no aparece en R ni en T, las variables cuantificadas pueden renombrarse como sigue:

$$\langle \oplus i \ : R.i : \ T.i \rangle = \langle \oplus j \ : R.(f.j) : \ T.(f.j) \rangle$$

La inversa en el rango considerado se denotará con f^{-1}. La propiedad de ser inversa puede escribirse como

$$\left\langle \forall i, j \ : R.i \wedge R.(f.j) : \ f.i = j \equiv i = f^{-1}.j \right\rangle$$

DEMOSTRACIÓN
Comenzamos la demostración con la expresión más complicada

$$
\begin{aligned}
& \langle \oplus j \ : R.(f.j) : \ T.(f.j) \rangle \\
= & \ \{ \text{ Rango unitario (introducción de la cuantificación sobre } i) \ \} \\
& \langle \oplus j \ : R.(f.j) : \ \langle \oplus i \ : i = f.j : \ T.i \rangle \rangle \\
= & \ \{ \text{ Anidado } \} \\
& \langle \oplus i, j \ : R.(f.j) \wedge i = f.j : \ T.i \rangle \\
= & \ \{ \text{ Leibniz } \} \\
& \langle \oplus i, j \ : R.i \wedge i = f.j : \ T.i \rangle \\
= & \ \{ \text{ Anidado } \} \\
& \langle \oplus i \ : R.i : \ \langle \oplus j \ : i = f.j : \ T.i \rangle \rangle \\
= & \ \{ \text{ Inversa } \} \\
& \left\langle \oplus i \ : R.i : \ \left\langle \oplus j \ : j = f^{-1} : \ T.i \right\rangle \right\rangle \\
= & \ \{ \text{ Rango unitario, } j \text{ no es libre en } T \ \} \\
& \langle \oplus i \ : R.i : \ T.i \rangle
\end{aligned}
$$

Teorema 1.40 (Separación de un término izquierdo)

$$\langle \oplus i \ : 0 \leqslant i < n+1 : \ T.i \rangle = T.0 \oplus \langle \oplus i \ : 0 \leqslant i < n : \ T.(i+1) \rangle$$

Teorema 1.41 (Separación de un término derecho)

$$\langle \oplus i \ : 0 \leqslant i < n+1 : \ T.i \rangle = \langle \oplus i \ : 0 \leqslant i < n : \ T.i \rangle \oplus T.n$$

Para el cuantificador universal hay una regla extra, cuya importancia radica en el hecho que provee un modo de "pasar del rango al término" y viceversa:

***Regla de intercambio*:** $\langle \forall i \ : R.i : \ T.i \rangle \ \equiv \ \langle \forall i \ : \mathsf{T} : \ \neg R.i \vee T.i \rangle$
$\equiv \ \langle \forall i \ :: \ R.i \Rightarrow T.i \rangle$.

Las cuantificaciones universal y existencial están vinculadas a través de dos reglas importantes, que son una generalización de las leyes de De Morgan:

Reglas de De Morgan: $\neg \langle \forall i \ : R.i : \ T.i \rangle \ \equiv \ \langle \exists i \ : R.i : \ \neg T.i \rangle$
$\neg \langle \exists i \ : R.i : \ T.i \rangle \ \equiv \ \langle \forall i \ : R.i : \ \neg T.i \rangle$.

Teorema 1.42 (Instanciación)

$$\langle \forall x \ :: \ f.x \rangle \Rightarrow f.Y$$

DEMOSTRACIÓN
Probemos su equivalente $\langle \forall x \ :: \ f.x \rangle \wedge f.Y \equiv \langle \forall x \ :: \ f.x \rangle$

$\langle \forall x :: f.x \rangle$
$\equiv$ { Rango T }
$\langle \forall x : \mathsf{T} : f.x \rangle$
$\equiv$ { Absorbente del $\vee$ }
$\langle \forall x : \mathsf{T} \cup x = Y : f.x \rangle$
$\equiv$ { Partición de rango }
$\langle \forall x : \mathsf{T} : f.x \rangle \wedge \langle \forall x : x = Y : f.x \rangle$
$\equiv$ { Rango T y Rango unitario (notar que valen las hipótesis) }
$\langle \forall x :: f.x \rangle \wedge f.Y$

Otras propiedades de los cuantificadores universales y existenciales son expuestas en la forma de metateoremas ya que su validez es demostrada utilizando el mismo esquema de demostración.

Metateorema 1.43 (Generalización)
P es un teorema si y solo si $\langle \forall i :: P \rangle$ es un teorema

DEMOSTRACIÓN
La demostración puede hacerse por doble implicación. Una de ellas es inmediata por la propiedad de instanciación. Para ver la otra mostraremos cómo transformar una demostración de P en una de $\langle \forall i :: P \rangle$.

Supongamos que tenemos una demostración de P de la siguiente forma:

P
$\equiv$ { Razón 1 }
P_1
$\vdots$
P_n
$\equiv$ { Razón $n+1$ }
T

Podemos construir entonces de manera mecánica la siguiente demostración de $\langle \forall i :: P \rangle$:

$\langle \forall i :: P \rangle$
$\equiv$ { Razón 1 }
$\langle \forall i :: P_0 \rangle$
$\vdots$
$\langle \forall i :: P_n \rangle$
$\equiv$ { Razón $n+1$ }
$\langle \forall i :: \mathsf{T} \rangle$
$\equiv$ { Término constante }
T

Metateorema 1.44 (Testigo)
Si k no es libre en P ni en Q entonces $\langle \exists i \ :: \ P \rangle \Rightarrow Q$ si y solo si $P(i := k) \Rightarrow Q$ es un teorema.

DEMOSTRACIÓN

$$
\begin{aligned}
& \langle \exists i \ :: \ P \rangle \Rightarrow Q \\
\equiv \ & \{ \text{ implicación } \} \\
& \neg \langle \exists i \ :: \ P \rangle \vee Q \\
\equiv \ & \{ \text{ de Morgan } \} \\
& \langle \forall i \ :: \ \neg P \rangle \vee Q \\
\equiv \ & \{ \text{ cambio de variables, } k \text{ no es libre en } P \ \} \\
& \langle \forall k \ :: \ \neg P(i := k) \rangle \vee Q \\
\equiv \ & \{ \text{ distributividad, } k \text{ no es libre en } P \ \} \\
& \langle \forall k \ :: \ \neg P(i := k) \vee Q \rangle \\
\equiv \ & \{ \text{ implicación } \} \\
& \langle \forall k \ :: \ P(i := k) \Rightarrow Q \rangle
\end{aligned}
$$

Aplicando el metateorema 1.43 de la cuantificación universal, vemos que la última línea es un teorema si y solo si $P(i := k) \Rightarrow Q$ lo es. □

Capítulo 2

Invariantes

En el capítulo anterior formalizamos el concepto de sistema de transiciones y vimos como estos pueden ser utilizados para denotar programas. Además generalizamos el concepto de predicados y transformadores sobre esta formalización. Vimos por último como los transformadores pueden usarse para denotar un paso de ejecución en la semántica operacional. En este capítulo veremos el concepto clásico de invariante [DS90] generalizado a sistemas de transiciones, significando propiedades que se cumplen durante múltiples pasos de ejecución de un sistema de transiciones. Para ello primero desarrollaremos la teoría de puntos fijos que será utilizada más adelante sobre transformadores de predicados permitiéndonos caracterizar distintas formas de invariancia.

2.1 Puntos Fijos

Antes de comenzar exponiendo la teoría de puntos fijos propiamente dicha desarrollaremos las estructuras algebraicas y sus propiedades para tal fin. Estas serán una extensión de las propiedades en las secciones 1.4.3 y 1.6.3 pero aplicadas a posets en general lo cual nos permitirá definir distintas condiciones de existencia de los puntos fijos de funciones.

2.1.1 Estructuras algebraicas y juntividad

Para definir las propiedades de juntividad definiremos estructuras un poco más simples que reticulados:

Definición 2.1 (Semireticulado)
Dado un poset $(P, \sqsubseteq)$ no vacío

- Será un $\sqcup$-*semireticulado* (semi-reticulado con supremo) si todo par de elementos tiene menor cota superior. Para todo par de elementos $x, y \in P$ esta será denotada por $x \sqcup y$.

- De forma dual, será un $\sqcap$-*semireticulado* (semi-reticulado con ínfimo) si todo par de elementos tiene mayor cota inferior. Para todo par de elementos $x, y \in P$ esta será denotada por $x \sqcap y$.

- Si el poset es $\sqcap$-semireticulado y además todo subconjunto no vacío tiene mayor cota inferior (ínfimo) entonces diremos que es un *$\sqcap$-semireticulado casi completo*. Claramente estas estructuras tienen menor elemento $\perp$ como el ínfimo de todo el conjunto P.

- De forma dual, si el poset es $\sqcup$-semireticulado y además todo subconjunto no vacío tiene menor cota superior (supremo) entonces diremos que es un *$\sqcup$-semireticulado casi completo*. Claramente estas estructuras tienen mayor elemento $\top$ como el supremo de todo el conjunto P.

Por ejemplo, el poset de los números naturales $(\mathbb{N}, \leqslant)$ es un reticulado ya que es un $\sqcup$-semireticulado y $\sqcap$-semireticulado, no es un reticulado completo ya que no tiene máximo elemento $\top$ pero es un $\sqcap$-semireticulado casi completo ya que todo subconjunto no vacío tiene ínfimo. Notar el conjunto vacío no tiene ínfimo en $\mathbb{N}$ ya que su conjunto de cotas inferiores es $\mathbb{N}$ y este no tiene máximo elemento (carece del ∞).

Otras estructuras necesarias para definir otros tipos de juntividades son los *cpos* y *co-cpos*:

Definición 2.2 (cpo y co-cpo)
Dado un poset $(P, \sqsubseteq)$ no vacío

- será *cpo* (conjunto parcialmente ordenado) si todo subconjunto dirigido (definición 1.9, pág. 14) tiene menor cota superior (supremo). Notar que un cpo tiene menor elemento $\perp$ ya que el conjunto vacío es dirigido.

- De forma dual, será *co-cpo* si todo subconjunto codirigido (definición 1.9) tiene mayor cota inferior (ínfimo). Notar que un co-cpo tiene mayor elemento $\top$ ya que el conjunto vacío es codirigido.

- Será *cpo local* si todo subconjunto dirigido y acotado superiormente tiene supremo.

- De forma dual, será *co-cpo local* si todo subconjunto codirigido y acotado inferiormente tiene ínfimo. $\square$

Por ejemplo, el reticulado de los números naturales $(\mathbb{N}, \leqslant)$ no es cpo ya que él mismo es un conjunto dirigido y no posee supremo (no posee elemento ∞) pero es un cpo local. Notar que estas últimas definiciones permiten extender la noción de continuidad dada en la definición 1.10 (pág. 14) a cpos y co-cpos. Más adelante explicitaremos la definición sobre estas nuevas estructuras.

Todo reticulado completo posee la propiedad de ser cpo y co-cpo ya que posee ínfimo y supremo para cualquier subconjunto. Por lo tanto estas estructuras son más débiles que los reticulados completos. Como veremos existe una relación entre semireticulados casi completos y reticulados completos:

Teorema 2.3
Dado un poset $(P, \sqsubseteq)$ no vacío, las siguientes proposiciones son equivalentes:

- $(P, \sqsubseteq)$ es reticulado completo.

- $(P, \sqsubseteq)$ tiene mayor elemento $\top$ y es $\sqcap$-semireticulado casi completo.

- $(P, \sqsubseteq)$ tiene menor elemento $\perp$ y es $\sqcup$-semireticulado casi completo.

Notar que a partir de este teorema no es necesario hacer la distinción entre $\sqcap$-semireticulado completo y $\sqcup$-semireticulado completo ya que las nociones son equivalentes. La demostraciones de este teorema pueden encontrarse en [DP90, teorema 2.16].

Existe además otra propiedad que relaciona cpos locales con semireticulados casi completos:

Teorema 2.4
Si el poset $(P, \sqsubseteq)$ es $\sqcap$-semireticulado casi completo entonces es cpo local. Y de forma dual, si es $\sqcup$-semireticulado casi completo entonces es co-cpo local. □

Esto quiere decir que la propiedad de ser semireticulado casi completo es más fuerte que la de ser cpo local. La demostración puede encontrarse en [MRS93].

A partir de estas estructuras generalizaremos los diferentes tipos de juntividad ya presentados en la sección 1.4.3 :

Definición 2.5 (juntividad)
Dado un poset $(P, \sqsubseteq)$ no vacío y una función $f : P \to P$.

- f es *monótona* si para todo par de elementos $x, y \in P$ se cumple:

$$x \sqsubseteq y \Rightarrow f.x \sqsubseteq f.y \ .$$

- Si $(P, \sqsubseteq)$ tiene menor elemento $\bot$, f es *estricta* si $f.\bot = \bot$.

- Si $(P, \sqsubseteq)$ tiene mayor elemento $\top$, f es *no interrumpible* si $f.\top = \top$.

- Si $(P, \sqsubseteq)$ es un $\sqcap$-semireticulado casi completo, f es *positivamente conjuntiva* si el ínfimo distribuye sobre subconjuntos no vacíos arbitrarios. Esto es, para todo conjunto de índices $I \subseteq \Gamma$ con $I \neq \varnothing$ y para todo $C : \Gamma \to P$ se cumple:

$$f.\langle \textstyle\bigsqcap i \ : I.i : \ C.i \rangle = \langle \textstyle\bigsqcap i \ : I.i : \ f.(C.i) \rangle \ .$$

- De forma dual, si $(P, \sqsubseteq)$ es un $\sqcup$-semireticulado casi completo, f es *positivamente disyuntiva* si para todo conjunto de índices $I \subseteq \Gamma$ con $I \neq \varnothing$ y para todo $C : \Gamma \to P$ se cumple:

$$f.\langle \textstyle\bigsqcup i \ : I.i : \ C.i \rangle = \langle \textstyle\bigsqcup i \ : I.i : \ f.(C.i) \rangle \ .$$

- Si $(P, \sqsubseteq)$ es un reticulado completo, f es *universalmente conjuntiva* si es no interrumpible y *positivamente conjuntiva* . O sea es conjuntiva para subconjuntos arbitrarios.

- Si $(P, \sqsubseteq)$ es un reticulado completo, f es *universalmente disyuntiva* si es estricta y *positivamente disyuntiva* . O sea es disyuntiva para subconjuntos arbitrarios.

- Si $(P, \sqsubseteq)$ es un cpo, f es $\sqcup$-*continua* si es disyuntiva sobre conjuntos dirigidos no vacíos. Esto es, para todo conjunto de índices $I \subseteq \Gamma$ con $I \neq \varnothing$ y para todo $C : \Gamma \to P$ tal que $\{C.i \mid i \in I\}$ es dirigido, se cumple:

$$f.\langle \textstyle\bigsqcup i \ : I.i : \ C.i \rangle = \langle \textstyle\bigsqcup i \ : I.i : \ f.(C.i) \rangle \ .$$

- De forma dual, si $(P, \sqsubseteq)$ es un co-cpo, f es $\sqcap$-*continua* si es conjuntiva sobre conjuntos codirigidos no vacíos. Esto es, para todo conjunto de índices $I \subseteq \Gamma$ con $I \neq \varnothing$ y para todo $C : \Gamma \to P$ tal que $\{C.i \mid i \in I\}$ es codirigido, se cumple:

$$f.\langle \sqcap i \ : I.i : \ C.i \rangle = \langle \sqcap i \ : I.i : \ f.(C.i) \rangle \ . \qquad \square$$

Como ya vimos es el capítulo anterior para el caso restringido de Pred y $\mathsf{Pred}_{\mathcal{L}}$ los diferentes tipos de juntividad pueden ser ordenados según su fortaleza:

Teorema 2.6 (Jerarquía de juntividades)
Sobre los dominios en que estén definidos los siguientes tipos de juntividad, se cumple que:

universalmente conjuntivo(disyuntivo)
$\Rightarrow$ positivamente conjuntivo(disyuntivo)
$\Rightarrow$ $\sqcap(\sqcup)$-continuo
$\Rightarrow$ monótono $\square$

Además vimos que WLP es universalmente conjuntivo por lo que es positivamente conjuntivo, $\sqcap$-continuo y monótono. Las propiedades duales se cumplen para el transformador SP.

2.1.2 Puntos fijos

Dado un poset $(P, \sqsubseteq)$ un punto fijo de una función $f : P \to P$ será un elemento $x \in P$ tal que $f.x = x$. Claramente no cualquier función definida sobre un poset posee punto fijo. Por ejemplo en el poset $(\mathbb{N}, \leqslant)$ la función $f.n \ \doteq \ n + 1$ no lo tiene.

Si una función en un poset posee puntos fijos, otra pregunta posible es sobre la existencia de de los menores y mayores entre ellos. Esto es, existe un elemento $x \in P$ tal que $f.x = x$ (es punto fijo) y $\langle \forall\, y \in P \ : f.y = y : \ x \sqsubseteq y \rangle$ para el caso del menor o $\langle \forall\, y \in P \ : f.y = y : \ y \sqsubseteq x \rangle$ para el caso del mayor punto fijo. En el caso que existan denotaremos $\mu.f$ y $\nu.f$ al menor y mayor punto fijo respectivamente. A continuación veremos algunos teoremas que aseguran su existencia bajo ciertas restricciones.

Teorema 2.7 (Knaster-Tarski)
Dado un poset $(P, \sqsubseteq)$ y una función monótona (definición 1.7) $f : P \to P$. Si $(P, \sqsubseteq)$ es un reticulado completo entonces existen el menor y mayor punto fijo y están caracterizados como

$$\begin{array}{lcll} \mu.f & = & \langle \sqcap x \in P \ : f.x \sqsubseteq x : \ x \rangle & \text{(caracterización } \mu) \\ \nu.f & = & \langle \sqcup x \in P \ : x \sqsubseteq f.x : \ x \rangle & \text{(caracterización } \nu) \end{array} \qquad \square$$

La demostración del teorema puede encontrarse en [BAW98, capítulo 19].

A un elemento $x \in P$ tal que $f.x \sqsubseteq x$ ($x \sqsubseteq f.x$) lo llamaremos pre(pos)-punto fijo. El teorema muestra que bajo sus condiciones el mayor y menor punto fijo existen y son el ínfimo y supremo de los pre-puntos fijos y los pos-puntos fijos respectivamente.

Como vimos en el capítulo anterior Pred y $\mathsf{Pred}_{\mathcal{L}}$ son reticulados completos, por lo cual puede aplicarse este teorema de existencia. De todas formas, la

caracterización de los puntos fijos como supremos o ínfimos de conjuntos, aunque útil al momento de demostrar sus propiedades, puede ser engorrosa si queremos obtenerlos efectivamente. Esto es debido a que los conjuntos de pre y pos-puntos fijos pueden ser muy grandes (incluso pueden llegar a ser no numerables). A continuación desarrollaremos un teorema que caracteriza los puntos fijos como límite de secuencias numerables.

Teorema 2.8
Dado un cpo $(P, \sqsubseteq)$ toda función $\sqcup$-continua $f : P \to P$ tiene un menor punto fijo dado por

$$\mu.f = \langle \bigsqcup n \in \mathbb{N} \ :: \ f^n.\bot \rangle \ .$$

De forma dual, dado un co-cpo $(P, \sqsubseteq)$ toda función $\sqcap$-continua $f : P \to P$ tiene un mayor punto fijo dado por

$$\nu.f = \langle \bigsqcap n \in \mathbb{N} \ :: \ f^n.\top \rangle \ . \qquad \square$$

Notar en el teorema que el supremo sobre el conjunto $\{f^n.\bot \mid n \in \mathbb{N}\}$ está bien definido ya que es dirigido (es una cadena). Esta propiedad puede demostrarse gracias la monotonía de la función f (continuo implica monótono por teorema 2.6). Lo mismo sucede en la parte dual del teorema. Cabe mencionar que para toda función monótona en general, el límite $\langle \bigsqcup n \in \gamma \ :: \ f^n.\bot \rangle$ existe para algún ordinal γ fijo en el poset [BAW98, capítulo 19]. Pidiendo continuidad de f podemos alcanzarlo simplemente con el primer ordinal $\mathbb{N}$.

En la sección anterior vimos que $(\mathbb{N}, \leqslant)$ es un $\sqcap$-semireticulado casi completo pero no es un cpo (aunque es un cpo local por teorema 2.4). La función $f : \mathbb{N} \to \mathbb{N}$ definida como $f.n \ \doteq \ n+1$ claramente es monótona y no posee punto fijo. Queda por investigar bajo que condiciones se puede asegurar la existencia de puntos fijos en semireticulados casi completos.

Teorema 2.9
Dado $(P, \sqsubseteq)$ un $\sqcap$-semireticulado casi completo y $f : P \to P$ una función monótona. Si existe un pre-punto fijo de f entonces tiene menor punto fijo dado por:

$$\mu.f = \langle \bigsqcap x \in P \ : f.x \sqsubseteq x : \ x \rangle \ .$$

De forma dual, si $(P, \sqsubseteq)$ es un $\sqcup$-semireticulado casi completo y f tiene al menos un pos-punto fijo, el mayor punto fijo está dado por:

$$\nu.f = \langle \bigsqcup x \in P \ : x \sqsubseteq f.x : \ x \rangle \ . \qquad \square$$

Por lo tanto podemos asegurar la existencia de puntos fijos en semireticulados casi completos únicamente si la función en cuestión tiene al menos un pre(pos)-punto fijo. Cabe agregar que si el poset es un cpo local el menor punto fijo también existe y es caracterizado por $\langle \bigsqcup n \in \gamma \ :: \ f^n.\bot \rangle$ para algún ordinal γ (la propiedad dual también se cumple). Las demostraciones de estas propiedades pueden encontrarse en [MRS93].

2.2 Invariantes

En esta sección definiremos distintos conceptos de invariantes como propiedades que se cumplen a lo largo de todos los pasos de ejecución de un sistema de transiciones. El primero estará relacionado con su semántica operacional (sección 1.7).

Definición 2.10 (Invariante)
Dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$, $P : \mathsf{Pred}_{\mathcal{L}}$ es un *invariante* si todas las configuraciones en todas la ejecuciones de TS satisfacen P:

$$\langle \forall \varrho, l, \sigma \ : \varrho \in [\![\mathsf{TS}]\!] \wedge (l, \sigma) \in \rho : \ P.l.\sigma \rangle \ .$$

Esta situación será denotada como $\mathsf{TS} \models \Box P$. Cuando se sobreentienda del contexto escribiremos simplemente $\Box P$. □

A continuación veremos algunas propiedades de los invariantes.

Propiedades 2.11 (Invariantes)
Dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ y dos predicados P, Q : $\mathsf{Pred}_{\mathcal{L}}$ se cumple:

1. $\Box \mathsf{True}$.
2. $\Box P \ \Rightarrow \ \Theta \subseteq P$.
3. $\Box P \wedge P \subseteq Q \ \Rightarrow \ \Box Q$.
4. $\Box P \wedge \Box Q \ \Rightarrow \ \Box (P \cap Q)$. □

Estas propiedades pueden ser demostradas fácilmente a partir de la definición anterior.

Probar que un predicado P es un invariante puede ser engorroso utilizando la semántica operacional. Afortunadamente podemos utilizar las semánticas WLP y SP para definir otro tipo de invariancia que, como veremos maś adelante, está relacionada con la anterior.

Definición 2.12 (Invariante inductivo)
Dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$, $\varphi : \mathsf{Pred}_{\mathcal{L}}$ es un *invariante inductivo* si se cumple:

1. $\Theta \subseteq \varphi$,
2. $\{\varphi\} \ \mathcal{T} \ \{\varphi\}$.

Si solo se cumple la condición 2 diremos que φ es un *invariante inductivo candidato.* □

Recordemos que $\{\varphi\} \ \mathcal{T} \ \{\varphi\}$ denota tanto a $\varphi \subseteq \mathrm{WLP}.\mathcal{T}.\varphi$ como a $\mathrm{SP}.\mathcal{T}.\varphi \subseteq \varphi$ (sección 1.6.3). Como puede verse, esta nueva definición caracteriza a un invariante como un predicado que se cumple inicialmente (condición 1) y se preserva por aplicación de los transformadores (condición 2). Si se cumple solo la última condición será solamente un invariante inductivo candidato.

De aquí en adelante usaremos letras griegas para denotar invariantes inductivos. El siguiente lema muestra la relación entre ambas definiciones de invariante.

Lema 2.13
Dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$, si $\varphi : \mathsf{Pred}_{\mathcal{L}}$ es un invariante inductivo entonces se cumple $\mathsf{TS} \models \Box \varphi$ (es invariante).

DEMOSTRACIÓN
Supongamos $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ y φ un invariante inductivo del mismo. Sea $\varrho \in [\![\mathsf{TS}]\!]$ (una ejecución de TS). Veremos que todas las configuraciones en ϱ cumplen φ. La prueba será por inducción en la longitud de ϱ.

Caso base

$$
\begin{array}{ll}
 & \varphi.l_0.\sigma_0 \\
 & \quad [\![(l_0, \sigma_0) = \varrho.0]\!] \\
\Leftarrow & \{\ \Theta \subseteq \varphi \text{ (por definición de invariante inductivo) }\} \\
 & \Theta.l_0.\sigma_0 \\
 & \quad [\![(l_0, \sigma_0) = \varrho.0]\!] \\
\Leftarrow & \{\ \varrho \text{ es ejecución (definición 1.26) }\} \\
 & \mathsf{T}
\end{array}
$$

Paso inductivo (Hipótesis inductiva $\varphi.l_i.\sigma_i$)

$$
\begin{array}{ll}
 & \varphi.l_{i+1}.\sigma_{i+1} \\
 & \quad [\![(l_{i+1}, \sigma_{i+1}) = \varrho.(i+1)]\!] \\
\Leftarrow & \{\ \text{Definición de invariante inductivo }\} \\
 & \text{SP}.\mathcal{T}.\varphi.l_{i+1}.\sigma_{i+1} \\
 & \quad [\![(l_{i+1}, \sigma_{i+1}) = \varrho.(i+1)]\!] \\
\Leftarrow & \{\ \text{Resultado en sección 1.7 }\} \\
 & \langle\, \exists\, l, \sigma\ : \varphi.l.\sigma :\ (l, \sigma) \rightsquigarrow (l_{i+1}, \sigma_{i+1})\, \rangle \\
 & \quad [\![(l_{i+1}, \sigma_{i+1}) = \varrho.(i+1)]\!] \\
\Leftarrow & \{\ \text{Instanciación } l\ \sigma \text{ con } \varrho.i\ \} \\
 & \varphi.l_i.\sigma_i \wedge (l_i, \sigma_i) \rightsquigarrow (l_{i+1}, \sigma_{i+1}) \\
 & \quad [\![(l_{i+1}, \sigma_{i+1}) = \varrho.(i+1) \wedge (l_i, \sigma_i) = \varrho.i]\!] \\
\Leftarrow & \{\ \text{Hipótesis inductiva y definición de ejecución 1.26 }\} \\
 & \mathsf{T}
\end{array}
$$

□

Las condiciones en la definición 2.12, que caracterizan un invariante inductivo, resultan más simples de verificar con la lógica que la de invariante a secas (definición 2.10) ya que se utiliza la semántica de transformadores en vez de la operacional. En este sentido, una pregunta posible es si nos podemos manejar únicamente con la definición de invariante inductivo para demostrar propiedades sobre los sistemas de transiciones. El próximo teorema responde esta pregunta.

Teorema 2.14 (Regla del invariante)
Dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$. Un predicado $P : \mathsf{Pred}_{\mathcal{L}}$ es invariante ($\mathsf{TS} \models \Box P$) si y solo si existe $\varphi : \mathsf{Pred}_{\mathcal{L}}$ tal que

1. $\Theta \subseteq \varphi$,
2. $\{\varphi\}\ \mathcal{T}\ \{\varphi\}$ y
3. $\varphi \subseteq P$,
   o sea φ es un invariante inductivo más fuerte que P.

Demostración
Demostraremos la doble implicación.

($\Leftarrow$) Como φ es un invariante inductivo, por lema 2.13 también es un invariante a secas ($\Box\varphi$). Por lo tanto, utilizando la propiedad 2.11.3 y la hipótesis $\varphi \subseteq P$ se demuestra que P es un invariante ($\Box P$).

($\Rightarrow$) Como $\mathsf{Pred}_{\mathcal{L}}$ es un reticulado completo, podemos definir

$$\varphi \doteq \left\langle \bigcup i \;: i \in \mathbb{N} :\; (\mathrm{SP}.\mathcal{T})^i.\Theta \right\rangle \;.$$

Demostraremos primero que φ así definido es un invariante inductivo:

$$
\begin{aligned}
& \varphi \\
= \;& \{ \text{ Definición } \} \\
& \left\langle \bigcup i \;: i \in \mathbb{N} :\; (\mathrm{SP}.\mathcal{T})^i.\Theta \right\rangle \\
= \;& \{ \text{ Partición de rango, rango unitario } \} \\
& (\mathrm{SP}.\mathcal{T})^0.\Theta \;\cup\; \left\langle \bigcup i \;: i \in \mathbb{N} :\; (\mathrm{SP}.\mathcal{T})^{i+1}.\Theta \right\rangle \\
= \;& \{ \text{ SP es universalmente disyuntiva } \} \\
& (\mathrm{SP}.\mathcal{T})^0.\Theta \;\cup\; \mathrm{SP}.\mathcal{T}.\left\langle \bigcup i \;: i \in \mathbb{N} :\; (\mathrm{SP}.\mathcal{T})^i.\Theta \right\rangle \\
= \;& \{ \text{ Definición de } \varphi \text{ y simplificación } \} \\
& \Theta \;\cup\; \mathrm{SP}.\mathcal{T}.\varphi
\end{aligned}
$$

Entonces, según la demostración

$$\Theta \subseteq \varphi \wedge \mathrm{SP}.\mathcal{T}.\varphi \subseteq \varphi$$

y por lo tanto φ es un invariante inductivo.
Queda por demostrar que φ así definida es más fuerte que P:

$$
\begin{aligned}
& \varphi \subseteq P \\
\equiv \;& \{ \text{ Definición de } \varphi \; \} \\
& \left\langle \bigcup i \;: i \in \mathbb{N} :\; (\mathrm{SP}.\mathcal{T})^i.\Theta \right\rangle \subseteq P \\
\equiv \;& \{ \text{ Propiedad de supremo en reticulados completos } \} \\
& \left\langle \forall i \;: i \in \mathbb{N} :\; (\mathrm{SP}.\mathcal{T})^i.\Theta \subseteq P \right\rangle
\end{aligned}
$$

Utilizando principalmente el teorema 1.29 (pág. 30) demostraremos el termino de la cuantificación universal $(\mathrm{SP}.\mathcal{T})^n.\Theta \subseteq P$ para todo $n \in \mathbb{N}$ o lo que es lo mismo $(\mathrm{SP}.\mathcal{T})^n.\Theta.l'.\sigma' \Rightarrow P.l'.\sigma'$ para toda configuración (l', σ'):

$$
\begin{aligned}
& (\mathrm{SP}.\mathcal{T})^n.\Theta.l'.\sigma' \Rightarrow P.l'.\sigma' \\
\equiv \;& \{ \text{ Teorema 1.29 } \} \\
& \left\langle \exists \varrho \;: \varrho \in [\![\mathsf{TS}]\!] \wedge n \in \mathsf{Dom}.\varrho :\; \varrho.n = (l', \sigma') \right\rangle \Rightarrow P.l'.\sigma' \\
\equiv \;& \{ \text{ Lógica de predicados } \} \\
& \left\langle \forall \varrho \;: \varrho \in [\![\mathsf{TS}]\!] \wedge n \in \mathsf{Dom}.\varrho :\; \varrho.n = (l', \sigma') \Rightarrow P.l'.\sigma' \right\rangle \\
\equiv \;& \{ \text{ Intercambio } \} \\
& \left\langle \forall \varrho \;: \varrho \in [\![\mathsf{TS}]\!] :\; n \in \mathsf{Dom}.\varrho \wedge \varrho.n = (l', \sigma') \Rightarrow P.l'.\sigma' \right\rangle
\end{aligned}
$$

$\Leftarrow$ { Instanciación universal, fortalecimiento de término }

$\langle \forall \varrho : \varrho \in [\![\mathsf{TS}]\!] : \langle \forall l, \sigma :: (l,\sigma) \in \varrho \Rightarrow P.l.\sigma \rangle \rangle$

$\equiv$ { Intercambio }

$\langle \forall \varrho : \varrho \in [\![\mathsf{TS}]\!] : \langle \forall l, \sigma : (l,\sigma) \in \varrho : P.l.\sigma \rangle \rangle$

$\equiv$ { Anidado }

$\langle \forall \varrho, l, \sigma : \varrho \in [\![\mathsf{TS}]\!] \wedge (l,\sigma) \in \varrho : P.l.\sigma \rangle$

$\equiv$ { Definición 2.10 con P invariante }

T

Por lo tanto φ es un invariante inductivo más fuerte que P y finaliza la prueba del teorema. □

El teorema sirve como regla para establecer si un predicado $P : \mathsf{Pred}_{\mathcal{L}}$ es un invariante: P es invariante si encuentro un invariante inductivo más fuerte. Esta regla es correcta y relativamente completa [BBM97, pág. 3]. A partir de este último teorema podemos formular una definición alternativa de invariante:

Definición 2.15 (Invariante)
Dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ un predicado P es *invariante* del mismo cuando exista un predicado φ tal que se cumple

1. φ es un invariante inductivo,

2. $\varphi \subseteq P$.

Si solo se cumple

1. φ es un invariante inductivo candidato y

2. $\varphi \subseteq P$,

entonces P será llamado *invariante candidato*. □

Un ejemplo de invariante inductivo para cualquier sistema de transiciones es el predicado True (también es un invariante a secas); este predicado cumple trivialmente las condiciones de la definición 2.12 pero no sirve de mucho al momento de verificar si un predicado arbitrario P es invariante. Esto es debido a que por la definición anterior debe verificarse $\mathsf{True} \subseteq P$ y por lo tanto lo único que podemos probar es que True es un invariante.

2.3 Invariantes y puntos fijos

Planteada la pregunta si un predicado P es un invariante, a partir de la última observación puede percibirse que el problema reside en la posibilidad de encontrar un invariante inductivo lo suficientemente fuerte. En esta sección veremos como se pueden caracterizar distintos tipos de invariantes inductivos según su fortaleza apoyándonos en la teoría de puntos fijos desarrollada en la sección 2.1.

2.3.1 Propagación hacia adelante

Según definición 2.15 si queremos probar la invariancia de predicado P podríamos intentar encontrar el predicado φ más fuerte tal que

1. $\Theta \subseteq \varphi$ y
2. $\mathrm{SP}.\mathcal{T}.\varphi \subseteq \varphi$

y con el mismo intentar probar la condición $\varphi \subseteq P$. Las dos primeras condiciones pueden conjugarse en una equivalente:

$$\Theta \cup \mathrm{SP}.\mathcal{T}.\varphi \ \subseteq \ \varphi \ ,$$

por lo tanto lo que debemos encontrar es el predicado φ más fuerte que cumpla esta última condición. Observando la parte izquierda de esta ecuación, definamos la función sobre el reticulado $(\mathsf{Pred}_{\mathcal{L}}, \subseteq)$

$$\mathcal{F}_{\mathcal{T},\Theta}.X \ \doteq \ \Theta \ \cup \ \mathrm{SP}.\mathcal{T}.X \ ,$$

de forma tal que la última ecuación pueda escribirse en la forma del pre-punto fijo

$$\mathcal{F}_{\mathcal{T},\Theta}.\varphi \subseteq \varphi \ .$$

Por lo tanto lo que queremos encontrar es el menor predicado tal que esta última ecuación se cumpla, es decir queremos encontrar el menor pre-punto fijo de la función $\mathcal{F}_{\mathcal{T},\Theta}$ en el reticulado. El resultado de la ecuación existe ya que $\mathsf{Pred}_{\mathcal{L}}$ es un reticulado completo y además por teorema 2.7 (Knaster-Tarski) coincide con el menor punto fijo de $\mathcal{F}_{\mathcal{T},\Theta}$:

$$\mu.\mathcal{F}_{\mathcal{T},\Theta} = \langle \bigcap X \ : \mathcal{F}_{\mathcal{T},\Theta}.X \subseteq X : \ X \rangle \ .$$

Por lo tanto el invariante inductivo φ que intentamos encontrar es justamente el menor punto fijo $\mu.\mathcal{F}_{\mathcal{T},\Theta}$. Solo resta ver si se cumple la condición $\mu.\mathcal{F}_{\mathcal{T},\Theta} \subseteq P$ para comprobar si P es un invariante. A partir de este desarrollo introduciremos la siguiente definición y teorema.

Definición 2.16 (Propagación hacia adelante)
Dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ definiremos el transformador $\mathcal{F}_{\mathcal{T},\Theta} : \mathsf{Pred}_{\mathcal{L}} \to \mathsf{Pred}_{\mathcal{L}}$ como

$$\mathcal{F}_{\mathcal{T},\Theta}.X \ \doteq \ \Theta \ \cup \ \mathrm{SP}.\mathcal{T}.X \ . \qquad \square$$

Este transformador es positivamente disyuntivo por lo que según teorema 2.6 es $\sqcup$-continuo y monótono. Aunque SP es universalmente disyuntivo el nuevo transformador no hereda esta propiedad ya que $\mathcal{F}_{\mathcal{T},\Theta}.\mathsf{False} = \Theta$ (no es estricto).

Lema 2.17
Dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$, $\mu.\mathcal{F}_{\mathcal{T},\Theta}$ es un invariante inductivo. □

DEMOSTRACIÓN

$\mu.\mathcal{F}_{\mathcal{T},\Theta}$ es pre-punto fijo de $\mathcal{F}_{\mathcal{T},\Theta}$.

$\equiv$ { Definición de punto fijo }

$\mathcal{F}_{\mathcal{T},\Theta}.(\mu.\mathcal{F}_{\mathcal{T},\Theta}) \subseteq \mu.\mathcal{F}_{\mathcal{T},\Theta}$

$\equiv$ { Definición de $\mathcal{F}_{\mathcal{T},\Theta}$ }

$\Theta \cup \text{SP}.\mathcal{T}.(\mu.\mathcal{F}_{\mathcal{T},\Theta}) \subseteq \mu.\mathcal{F}_{\mathcal{T},\Theta}$

$\equiv$ { Lógica de predicados }

$\Theta \subseteq \mu.\mathcal{F}_{\mathcal{T},\Theta}$

$\wedge$

$\text{SP}.\mathcal{T}.(\mu.\mathcal{F}_{\mathcal{T},\Theta}) \subseteq \mu.\mathcal{F}_{\mathcal{T},\Theta}$

$\equiv$ { Definición 2.12 }

$\mu.\mathcal{F}_{\mathcal{T},\Theta}$ es un invariante inductivo □

Lema 2.18
Dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$, $\mu.\mathcal{F}_{\mathcal{T},\Theta}$ es el menor invariante inductivo. □

DEMOSTRACIÓN
Sea φ un invariante inductivo del sistema. Entonces $\Theta \subseteq \varphi$ y $\text{SP}.\mathcal{T}\varphi \subseteq \varphi$. Por lo tanto φ es un pre-punto fijo de $\mathcal{F}_{\mathcal{T},\Theta}$. Como $\mu.\mathcal{F}_{\mathcal{T},\Theta}$ es el mínimo de los pre-puntos fijos entonces es menor que φ. □

Teorema 2.19
Dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$, un predicado P sobre el mismo es invariante si y solo si se cumple $\mu.\mathcal{F}_{\mathcal{T},\Theta} \subseteq P$. □

DEMOSTRACIÓN
La demostración de este teorema se deriva de los lemas anteriores y el teorema 2.14. □

Notar que según este teorema $\mu.\mathcal{F}_{\mathcal{T},\Theta}$ es el invariante más fuerte del sistema de transiciones.

La definición de $\mu.\mathcal{F}_{\mathcal{T},\Theta}$ como el ínfimo de los pre-puntos fijos puede ser engorrosa al momento de intentar calcularlo ya que al menos habría que obtener todos los pre-puntos fijos del transformador $\mathcal{F}_{\mathcal{T},\Theta}$. Pero, ya que $\mathsf{Pred}_{\mathcal{L}}$ es un cpo podemos aplicar el teorema 2.8 y presentarlo como límite de una secuencia numerable de predicados:

$$\mu.\mathcal{F}_{\mathcal{T},\Theta} = \left\langle \bigcup\ i\ : i \in \mathbb{N} :\ \mathcal{F}^{i}_{\mathcal{T},\Theta}.\mathsf{False} \right\rangle \ .$$

Vamos a denominar la secuencia en cuestión como $\varphi_0, \varphi_1, \cdots$ definida como $\varphi_n \doteq \mathcal{F}^{n}_{\mathcal{T},\Theta}.\mathsf{False}$. Ver que con esta definición $\varphi_{n+1} = \mathcal{F}_{\mathcal{T},\Theta}.\varphi_n$. Mostraremos a partir de ella cual es el significado de aquel límite. El primer elemento de la secuencia es el mínimo elemento False del reticulado $(\mathsf{Pred}_{\mathcal{L}}, \subseteq)$. Veamos el segundo:

φ_1

$=$ { Definición de φ_1 }

$\mathcal{F}_{\mathcal{T},\Theta}.\mathsf{False}$

$=$ { Definición de $\mathcal{F}_{\mathcal{T},\Theta}$ }

$\Theta \cup \text{SP}.\mathcal{T}.\mathsf{False}$

$= \{$ SP es estricto $\}$

$\Theta \cup \mathsf{False}$

$= \{$ False es el menor elemento del reticulado $\}$

Θ .

Por lo tanto el segundo elemento de la secuencia es el conjunto de configuraciones iniciales Θ. El tercer elemento es claramente $\varphi_2 = \Theta \cup \text{SP}.\mathcal{T}.\Theta$ lo cual nos dice que este resultado es el conjunto de configuraciones iniciales más las alcanzables después de un paso de ejecución del sistema de transiciones. Veamos el cuarto elemento de la secuencia:

φ_3

$= \{$ Definición de φ_3 $\}$

$\mathcal{F}_{\mathcal{T},\Theta}.\varphi_2$

$= \{$ Definición de $\mathcal{F}_{\mathcal{T},\Theta}$ $\}$

$\Theta \cup \text{SP}.\mathcal{T}.\varphi_2$

$= \{$ Definición de φ_2 $\}$

$\Theta \cup \text{SP}.\mathcal{T}.(\Theta \cup \text{SP}.\mathcal{T}.\Theta)$

$= \{$ Disyuntividad de SP $\}$

$\Theta \cup \text{SP}.\mathcal{T}.\Theta \cup (\text{SP}.\mathcal{T})^2.\Theta$.

Con lo cual φ_3 es conjunto de configuraciones iniciales más las alcanzables después de uno o dos pasos de ejecución del sistema de transiciones (teorema 1.29, pág 30). Puede demostrarse fácilmente por inducción que

$$\varphi_{n+1} = \Theta \cup \left\langle \bigcup i \,:\, 0 < i \leqslant n \,:\, (\text{SP}.\mathcal{T})^i.\Theta \right\rangle$$

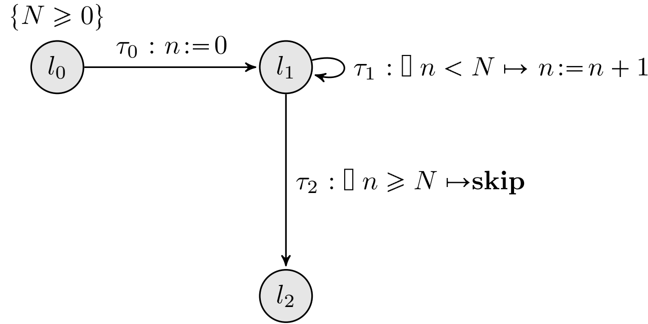
es decir φ_{n+1} es el conjunto de configuraciones iniciales más las alcanzables durante n pasos de ejecución. Además, a partir de esta ecuación, la secuencia forma una cadena ascendente comenzando desde False:

$$\underbrace{\mathsf{False}}_{\varphi^0} \subseteq \underbrace{\mathcal{F}_{\mathcal{T},\Theta}(\varphi_0)}_{\varphi^1} \subseteq \underbrace{\mathcal{F}_{\mathcal{T},\Theta}(\varphi_1)}_{\varphi^2} \subseteq \underbrace{\mathcal{F}_{\mathcal{T},\Theta}(\varphi_2)}_{\varphi^3} \subseteq \cdots$$

El límite de esta secuencia existe (es $\mu.\mathcal{F}_{\mathcal{T},\Theta}$) y coincide el conjunto de configuraciones alcanzables desde Θ en cualquier número de iteraciones del sistema de transiciones.

Ejemplo 2.20
Bajo esta perspectiva analicemos el sistema de transiciones 2.1 correspondiente al programa 1.1 (pág. 18) formulado en el capítulo anterior:

Grafo de transiciones 2.1 Programa 1.1



El predicado φ_0 es el mínimo del reticulado False o utilizando la notación de listas $\varphi_0 = [\mathsf{false}, \mathsf{false}, \mathsf{false}]$. Obtengamos el siguiente predicado de la cadena:

φ_1

$=$ { Definición de φ_1 }

$\mathcal{F}_{\mathcal{T},\Theta}.\mathsf{False}$

$=$ { Definición de $\mathcal{F}_{\mathcal{T},\Theta}$ }

$\Theta \cup \mathrm{SP}.\mathcal{T}.\mathsf{False}$

$=$ { SP es estricto }

$\Theta \cup \mathsf{False}$

$=$ { False es mínimo del reticulado }

Θ .

Por lo tanto $\varphi_1 = \Theta$. Veamos el siguiente elemento de la cadena:

φ_2

$=$ { Definición de φ_2 y $\mathcal{F}_{\mathcal{T},\Theta}$ }

$\Theta \cup \mathrm{SP}.\mathcal{T}.\varphi_1$

$=$ { Resultado anterior sobre φ_1 }

$\Theta \cup \mathrm{SP}.\mathcal{T}.\Theta$

$=$ { Notación en $\mathsf{Pred}_{\mathcal{L}}$ como lista. Definición de SP (ver ejemplo 1.17) }

$[N \geqslant 0, \mathsf{false}, \mathsf{false}] \cup [\mathsf{false}, \mathrm{sp}.\tau_0.(N \geqslant 0) \cup \mathrm{sp}.\tau_1.\mathsf{false}, \mathsf{false}]$

$=$ { Definición de sp. sp es estricto }

$[N \geqslant 0, \mathsf{false}, \mathsf{false}] \cup [\mathsf{false}, (N \geqslant 0 \wedge n = 0) \cup \mathsf{false}, \mathsf{false}]$

$=$ { $\sqcup$ en Pred y $\mathsf{Pred}_{\mathcal{L}}$. Aritmética }

$[N \geqslant 0, n = 0 \wedge n \geqslant N, \mathsf{false}]$

Este resultado muestra que φ_2 denota las configuraciones que se alcanzan después ejecutar la sentencia $n := 0$ (el primer paso de ejecución) más las configuraciones iniciales en Θ. Calculemos ahora φ_3:

$$
\begin{aligned}
& \varphi_3 \\
= & \ \{ \text{ Definición de } \varphi_3 \text{ y } \mathcal{F}_{\mathcal{T},\Theta} \ \} \\
& \Theta \cup \mathrm{SP}.\mathcal{T}.\varphi_2 \\
= & \ \{ \text{ Resultado anterior sobre } \varphi_2 \ \} \\
& \Theta \cup \mathrm{SP}.\mathcal{T}.[N \geqslant 0,\, N \geqslant 0 \wedge n = 0,\, \mathsf{false}] \\
= & \ \{ \text{ Notación en } \mathsf{Pred}_{\mathcal{L}} \text{ como lista. Definición de SP } \} \\
& [N \geqslant 0,\, \mathsf{false},\, \mathsf{false}] \cup \\
& [\mathsf{false},\, \mathrm{sp}.\tau_0.(N \geqslant 0) \cup \mathrm{sp}.\tau_1.(N \geqslant 0 \wedge n = 0),\, \mathrm{sp}.\tau_2.(N \geqslant 0 \wedge n = 0)] \\
= & \ \{ \text{ Definición de sp } \} \\
& [N \geqslant 0,\, \mathsf{false},\, \mathsf{false}] \cup \\
& [\mathsf{false},\, (N \geqslant 0 \wedge n = 0) \cup (N > 0 \wedge n = 1),\, N = 0 \wedge n = 0] \\
= & \ \{ \text{ Lógica de predicados, aritmética } \} \\
& [N \geqslant 0,\, \mathsf{false},\, \mathsf{false}] \cup \\
& [\mathsf{false},\, 0 \leqslant n < 2 \wedge n \leqslant N,\, n = 0 \wedge n = N] \\
= & \ \{ \sqcup \text{ en } \mathsf{Pred}_{\mathcal{L}} \ \} \\
& [N \geqslant 0,\, 0 \leqslant n < 2 \wedge n \leqslant N,\, n = 0 \wedge n = N]
\end{aligned}
$$

Este resultado muestra que φ_3 representa las configuraciones en el estado inicial más las alcanzables durante dos iteraciones del bucle. Por inducción se puede probar que

$$\varphi_{i+2} = [N \geqslant 0,\, 0 \leqslant n < i + 1 \wedge n \leqslant N,\, 0 \leqslant n < i \wedge n = N] \ .$$

Claramente este conjunto enumerado de fórmulas forma una cadena ascendente en $\mathsf{Pred}_{\mathcal{L}}$. Calculando el límite $\langle \bigcup i \,:\, i \in \mathbb{N} :\ \varphi_i \rangle$ obtenemos el menor punto fijo

$$\mu.\mathcal{F}_{\mathcal{T},\Theta} = [N \geqslant 0,\, 0 \leqslant n \leqslant N,\, 0 \leqslant n = N] \ .$$

Este resultado muestra que en la locación l_1 el programa 2.1 tiene como invariante más fuerte a $0 \leqslant n \leqslant N$ y en la locación l_2 a $0 \leqslant n = N$. De esta última observación y el teorema 2.19 se deduce que cuando el programa termina se cumple $n = N$.

Por último, con este resultado podemos anotar el programa 1.1 (pág. 18) con el nuevo invariante:

Programa 2.1 Programa 1.1 anotado

l_0: $\{N \geqslant 0\}$
  $n := 0;$
l_1: $\{0 \leqslant n \leqslant N\}$
  do $n < N \mapsto$
    $n := n + 1$
  od
l_2: $\{n = N\}$

2.3.2 Propagación hacia atrás

Otra estrategia para probar la invariancia de un predicado puede derivarse tomando la definición 2.15 (pág. 47) desde otra perspectiva. Dado un predicado P intentaremos encontrar un predicado ϕ tal que verifique

1. $\phi \subseteq \text{WLP}.\mathcal{T}.\phi$ y
2. $\phi \subseteq P$.

Para corroborar si P es un invariante restaría verificar $\Theta \subseteq \phi$. Sin esta condición ϕ es solo un invariante inductivo candidato. Notar que esta condición será necesaria si ϕ es el predicado más débil que verifica las condiciones 1 y 2. Las mismas pueden conjugarse en una equivalente:

$$\phi \;\subseteq\; P \cap \text{WLP}.\mathcal{T}.\phi \;\;,$$

por lo tanto lo que debemos encontrar es el predicado ϕ más débil que cumpla esta última ecuación. A partir de la parte derecha, definamos la función sobre el reticulado $(\mathsf{Pred}_{\mathcal{L}}, \subseteq)$

$$\mathcal{B}_{\mathcal{T},P}.X \;\doteq\; P \;\cap\; \text{WLP}.\mathcal{T}.X \;\;,$$

de forma tal que la última ecuación puede escribirse como

$$\phi \subseteq \mathcal{B}_{\mathcal{T},P}.\phi \;\;.$$

Como $\mathsf{Pred}_{\mathcal{L}}$ es un reticulado completo, haciendo el análisis dual del transformador de propagación hacia adelante podemos deducir que el máximo punto fijo de la ecuación existe y es el máximo de los pos-puntos fijos de $\mathcal{B}_{\mathcal{T},P}$:

$$\nu.\mathcal{B}_{\mathcal{T},P} = \langle \bigcup X \;: X \subseteq \mathcal{B}_{\mathcal{T},P}.X :\; X \rangle \;\;.$$

por lo tanto el mismo es el invariante candidato que deseamos encontrar. A partir de este desarrollo introduciremos la siguiente definición y teorema.

Definición 2.21 (Propagación hacia atrás)
Dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ y un predicado $P : \mathsf{Pred}_\mathcal{L}$ sobre el mismo definiremos el transformador $\mathcal{B}_{\mathcal{T},P} : \mathsf{Pred}_\mathcal{L} \to \mathsf{Pred}_\mathcal{L}$ como

$$\mathcal{B}_{\mathcal{T},P}.X \doteq P \cap \text{WLP}.\mathcal{T}.X \ . \qquad \square$$

Este transformador es positivamente conjuntivo por lo que según teorema 2.6 es $\sqcap$-continuo y monótono. Aunque WLP es universalmente conjuntivo $\mathcal{B}_{\mathcal{T},P}$ no hereda esta propiedad ya que $\mathcal{B}_{\mathcal{T},P}.\mathsf{True} = P$ (es interrumpible).

El siguiente lema muestra que el máximo punto fijo $\nu.\mathcal{B}_{\mathcal{T},P}$ es solo un invariante inductivo candidato.

Lema 2.22
Dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$, $\nu.\mathcal{B}_{\mathcal{T},P}$ es un invariante inductivo candidato. □

Demostración

$\nu.\mathcal{B}_{\mathcal{T},P}$ es pos-punto fijo de $\mathcal{B}_{\mathcal{T},P}$.

$\equiv$ { Definición de punto fijo }

$\nu.\mathcal{B}_{\mathcal{T},P} \subseteq \mathcal{B}_{\mathcal{T},P}.(\nu.\mathcal{B}_{\mathcal{T},P})$

$\equiv$ { Definición de $\mathcal{B}_{\mathcal{T},P}$ }

$\nu.\mathcal{B}_{\mathcal{T},P} \subseteq P \cap \text{WLP}.\mathcal{T}.(\nu.\mathcal{B}_{\mathcal{T},P})$

$\Rightarrow$ { Lógica de predicados }

$\nu.\mathcal{B}_{\mathcal{T},P} \subseteq \text{WLP}.\mathcal{T}.(\nu.\mathcal{B}_{\mathcal{T},P})$

$\Rightarrow$ { Definición 2.12 }

$\nu.\mathcal{B}_{\mathcal{T},P}$ es un invariante inductivo candidato. □

Además, este invariante inductivo candidato es el mayor (más débil) de los incluidos en la propiedad P (más fuertes que P):

Lema 2.23
Dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$, $\nu.\mathcal{B}_{\mathcal{T},P}$ es el mayor invariante inductivo candidato más fuerte que P. □

Demostración
Sea ϕ un invariante inductivo candidato del sistema tal que $\phi \subseteq P$. Por ser invariante inductivo candidato se cumple $\phi \subseteq \text{WLP}.\mathcal{T}\phi$. Por lo tanto ϕ es un pos-punto fijo de $\mathcal{B}_{\mathcal{T},P}$ y como $\nu.\mathcal{B}_{\mathcal{T},P}$ es el máximo de los pos-punto fijos entonces es mayor que φ. □

Este último lema junto con la definición de invariante 2.15 (pág. 47) muestran que a partir de $\nu.\mathcal{B}_{\mathcal{T},P}$ solo se puede deducir que P es un invariante candidato. El siguiente teorema agrega la condición necesaria para que el predicado sea un invariante:

Teorema 2.24
Dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$, un predicado P es invariante del mismo si y solo si se cumple $\Theta \subseteq \nu.\mathcal{B}_{\mathcal{T},P}$. □

Demostración
La demostración de este teorema se deriva de los lemas anteriores y el teorema 2.14. □

Notar que a partir de este teorema $\nu.\mathcal{B}_{\mathcal{T},P}$ puede ser visto como el conjunto de estados iniciales más débil a partir del cual el predicado P es invariante.

De igual manera que con el mínimo punto fijo $\mu.\mathcal{F}_{\mathcal{T},\Theta}$ el máximo punto fijo $\nu.\mathcal{B}_{\mathcal{T},P}$ puede escribirse como el límite de una cadena ya que $\mathsf{Pred}_{\mathcal{L}}$ es un co-cpo:

$$\mu.\mathcal{B}_{\mathcal{T},P} = \left\langle \bigcap i \ : i \in \mathbb{N} : \ \mathcal{B}^i_{\mathcal{T},P}.\mathsf{True} \right\rangle \ .$$

Vamos a denominar la secuencia en cuestión como $\phi_0, \phi_1, \cdots$ donde definimos $\phi_n \doteq \mathcal{B}^n_{\mathcal{T},P}.\mathsf{True}$. Ver que con esta definición $\phi_{n+1} = \mathcal{B}_{\mathcal{T},P}.\phi_n$. Mostraremos a partir de ella cual es el significado de aquel límite. El primer elemento de la secuencia es el mínimo elemento True. Veamos el segundo:

ϕ_1
$=$ { Definición de ϕ_1 }
$\mathcal{B}_{\mathcal{T},P}.\mathsf{True}$
$=$ { Definición de $\mathcal{B}_{\mathcal{T},P}$ }
$P \cap \mathrm{WLP}.\mathcal{T}.\mathsf{True}$
$=$ { WLP es no interrumpible }
$P \cap \mathsf{True}$
$=$ { True es el mayor elemento del reticulado }
P

Por lo tanto el segundo elemento de la secuencia es el conjunto de configuraciones iniciales P. El tercer elemento es $\phi_2 = P \cap \mathrm{WLP}.\mathcal{T}.P$ lo cual nos muestra que es el conjunto de configuraciones que cumplen P y desde las cuales si el programa termina lo hace a alguna configuración en P. El cuarto elemento de la secuencia es:

ϕ_3
$=$ { Definición de ϕ_3 }
$\mathcal{B}_{\mathcal{T},P}.\phi_2$
$=$ { Definición de $\mathcal{B}_{\mathcal{T},P}$ }
$P \cap \mathrm{WLP}.\mathcal{T}.\phi_2$
$=$ { Definición de ϕ_2 }
$P \cap \mathrm{WLP}.\mathcal{T}.(P \cap \mathrm{WLP}.\mathcal{T}.P)$
$=$ { Conjuntividad de WLP }
$P \cap \mathrm{WLP}.\mathcal{T}.P \cap (\mathrm{WLP}.\mathcal{T})^2.P$

Con lo cual ϕ_3 es conjunto de configuraciones que cumplen P y desde las cuales si el programa termina en una o dos iteraciones lo hace hacia alguna en P.

Puede demostrarse por inducción que

$$\phi_{n+1} = P \cap \left\langle \bigcap i \ : 0 < i \leqslant n : \ (\mathrm{WLP}.\mathcal{T})^i.P \right\rangle$$

por lo tanto ϕ_{n+1} es el conjunto de configuraciones que cumplen P y desde las cuales si el programa termina durante n pasos de ejecución lo hace a alguna en P. Como $\phi_n = \mathcal{B}^n_{\mathcal{T},P}.\mathsf{True}$ esta misma ecuación puede escribirse como

$$\mathcal{B}^n_{\mathcal{T},P}.\mathsf{True} = \left\langle \bigcap i \ : 0 \leqslant i < n : \ (\mathrm{WLP}.\mathcal{T})^i.P \right\rangle \ .$$

A partir de estas ecuaciones se puede ver que la secuencia forma una cadena descendente comenzando desde True:

$$\underbrace{\mathsf{True}}_{\phi^0} \supseteq \underbrace{\mathcal{B}_{\mathcal{T},P}(\phi_0)}_{\phi^1} \supseteq \underbrace{\mathcal{B}_{\mathcal{T},P}(\phi_1)}_{\phi^2} \supseteq \underbrace{\mathcal{B}_{\mathcal{T},P}(\phi_2)}_{\phi^3} \supseteq \cdots$$

El límite de esta secuencia existe y es el conjunto de configuraciones desde las cuales si el programa termina en cualquier número de iteraciones lo hace hacia alguna en P.

Cabe agregar que como los elementos de la secuencia son mayores al invariante inductivo candidato $\nu.\mathcal{B}_{\mathcal{T},P}$, por definición 2.15 (pág. 47) estos son invariantes candidatos. Solo si se verifica $\Theta \subseteq \nu.\mathcal{B}_{\mathcal{T},P}$ serán además invariantes a secas (no necesariamente inductivos).

Ejemplo 2.25
Bajo esta nueva perspectiva analicemos el mismo sistema de transiciones 2.1 (pág. 51). En este ejemplo intentaremos demostrar que la propiedad $n = N$ se cumple al finalizar el programa, es decir $P \doteq [\mathsf{True}, \mathsf{True}, n = N]$ es invariante del programa ($n = N$ se cumple en l_2).

En este caso el predicado ϕ_0 es el máximo del reticulado True o utilizando la notación de listas $\phi_0 = [\mathsf{true}, \mathsf{true}, \mathsf{true}]$. El siguiente elemento ϕ_1 de la cadena es P como vimos en el desarrollo anterior. Calculemos ϕ_2:

$$\begin{aligned}
& \phi_2 \\
= & \ \{ \text{ Definición de } \phi_2 \text{ y } \mathcal{B}_{\mathcal{T},P} \ \} \\
& P \cap \text{WLP}.\mathcal{T}.\phi_1 \\
= & \ \{ \text{ Resultado anterior sobre } \phi_1 \ \} \\
& P \cap \text{WLP}.\mathcal{T}.P \\
= & \ \{ \text{ Notación en } \mathsf{Pred}_{\mathcal{L}} \text{ como lista. Definición de WLP } \} \\
& [\, \mathsf{true},\ \mathsf{true},\ n = N \,] \cap [\, \mathsf{true},\ \text{wlp}.\tau_1.\mathsf{true} \cap \text{wlp}.\tau_2.(n = N),\ \mathsf{true} \,] \\
= & \ \{ \text{ Definición de wlp. wlp es no interrumpible } \} \\
& [\, \mathsf{true},\ \mathsf{true},\ n = N \,] \cap [\, \mathsf{true},\ \mathsf{true} \cap n \geqslant N \Rightarrow n = N,\ \mathsf{true} \,] \\
= & \ \{ \sqcap \text{ en } \mathsf{Pred} \text{ y } \mathsf{Pred}_{\mathcal{L}}\text{. Aritmética } \} \\
& [\, \mathsf{true},\ n \leqslant N,\ n = N \,]
\end{aligned}$$

Este resultado muestra que ϕ_2 son las configuraciones desde las cuales durante un paso de ejecución se alcanza el conjunto de estados $n = N$. Calculemos ahora ϕ_3:

$$\begin{aligned}
& \phi_3 \\
= & \ \{ \text{ Definición de } \phi_3 \text{ y } \mathcal{B}_{\mathcal{T},P} \ \} \\
& P \cap \text{WLP}.\mathcal{T}.\phi_2 \\
= & \ \{ \text{ Resultado anterior sobre } \phi_2 \ \} \\
& P \cap \text{WLP}.\mathcal{T}.[\, \mathsf{true},\ n \leqslant N,\ n = N \,] \\
= & \ \{ \text{ Notación en } \mathsf{Pred}_{\mathcal{L}} \text{ como lista. Definición de WLP } \} \\
& [\, \mathsf{true},\ \mathsf{true},\ n = N \,] \cap \\
& [\, \text{wlp}.\tau_0.(n \leqslant N),\ \text{wlp}.\tau_1.(n \leqslant N) \cap \text{wlp}.\tau_2.(n = N),\ \mathsf{true} \,]
\end{aligned}$$

$= \{$ Definición de wlp $\}$
$[\,\mathsf{true},\, \mathsf{true},\, n = N\,] \cap$
$[\,N \geqslant 0,\, n < N \Rightarrow n+1 \leqslant N \;\cap\; n \leqslant N,\, \mathsf{true}\,]$
$= \{$ Lógica de predicados, aritmética $\}$
$[\,\mathsf{true},\, \mathsf{true},\, n = N\,] \cap$
$[\,N \geqslant 0,\, n \leqslant N,\, \mathsf{true}\,]$
$= \{\ \sqcap$ en $\mathsf{Pred}_{\mathcal{L}}\ \}$
$[\,N \geqslant 0,\, n \leqslant N,\, n = N\,]$

Este resultado muestra que si comienzo el programa en alguna configuración $N \geqslant 0$ si el programa termina en tres pasos lo hace en P. Calculemos ahora ϕ_4:

ϕ_4
$= \{$ Definición de ϕ_4 y $\mathcal{B}_{\mathcal{T},P}$ $\}$
$P \cap \mathrm{WLP}.\mathcal{T}.\phi_3$
$= \{$ Resultado anterior sobre ϕ_3 $\}$
$P \cap \mathrm{WLP}.\mathcal{T}.[\,N \geqslant 0,\, n \leqslant N,\, n = N\,]$
$= \{$ Notación en $\mathsf{Pred}_{\mathcal{L}}$ como lista. Definición de WLP $\}$
$[\,\mathsf{true},\, \mathsf{true},\, n = N\,] \cap$
$[\,\mathrm{wlp}.\tau_0.(n \leqslant N),\, \mathrm{wlp}.\tau_1.(n \leqslant N) \cap \mathrm{wlp}.\tau_2.(n = N),\, \mathsf{true}\,]$
$= \{$ Demostración anterior $\}$
$[\,N \geqslant 0,\, n \leqslant N,\, n = N\,]$

Por lo tanto $\phi_4 = \phi_3$ con lo cual todos lo elementos de la cadena son iguales a ϕ_3 y hemos alcanzado el punto fijo $\nu.\mathcal{B}_{\mathcal{T},P}$:

$$\nu.\mathcal{B}_{\mathcal{T},P} = [\,N \geqslant 0,\, n \leqslant N,\, n = N\,]\ .$$

Para comprobar que P es un invariante resta verificar $\Theta \subseteq \nu.\mathcal{B}_{\mathcal{T},P}$ (sin esto P es solo un invariante candidato) o en notación de listas

$$[\,N \geqslant 0,\, \mathsf{false},\, \mathsf{false}] \subseteq [\,N \geqslant 0,\, n \leqslant N,\, n = N\,]$$

lo cual es claramente verdadero.

Por último, con este resultado podemos anotar el programa 1.1 (pág. 18) con el nuevo invariante:

Programa 2.2 Programa 1.1 anotado

l_0: $\{N \geqslant 0\}$
  $n := 0;$
l_1: $\{n \leqslant N\}$
  do $n < N \mapsto$
    $n := n + 1$
  od
l_2: $\{n = N\}$

Cabe remarcar que para realizar esta anotación no fue necesario calcular el límite de una cadena infinita, como lo hicimos en el ejemplo 2.20 (pág. 50), ya que en ϕ_3 se alcanza el punto fijo.

2.3.3 Propiedades

Como hemos visto, dado un sistema de transiciones, el mínimo punto fijo $\mu.\mathcal{F}_{\mathcal{T},\Theta}$ caracteriza de forma exacta las configuraciones alcanzables, mientras que dado un predicado P el máximo punto fijo $\nu.\mathcal{B}_{\mathcal{T},P}$ colecta la mínima información necesaria para probarlo. Con esta observación intuitiva en mente enunciaremos el siguiente teorema sobre la relación existente entre estos puntos fijos.

Teorema 2.26
Dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ y un predicado P sobre el mismo las siguientes proposiciones son equivalentes:

1. $\Theta \subseteq \nu.\mathcal{B}_{\mathcal{T},P}$.
2. $\mu.\mathcal{F}_{\mathcal{T},\Theta} \subseteq P$.
3. $\mu.\mathcal{F}_{\mathcal{T},\Theta} \subseteq \nu.\mathcal{B}_{\mathcal{T},P}$
4. $\Box P$

DEMOSTRACIÓN
(1 ⇒ 3) $\quad \Theta \subseteq \nu.\mathcal{B}_{\mathcal{T},P}$
$\Rightarrow$ { Lema 2.22 }
$\nu.\mathcal{B}_{\mathcal{T},P}$ es un invariante inductivo.
$\Rightarrow$ { Lema 2.18 }
$\mu.\mathcal{F}_{\mathcal{T},\Theta} \subseteq \nu.\mathcal{B}_{\mathcal{T},P}$

(3 ⇒ 2) Por lemma 2.23 $\nu.\mathcal{B}_{\mathcal{T},P} \subseteq P$. Entonces por hipótesis $\mu.\mathcal{F}_{\mathcal{T},\Theta} \subseteq P$.

(2 ≡ 4) Por teorema 2.19.

(4 ≡ 1) Por teorema 2.24.

La relación entre los distintos predicados es mostrada en el diagrama de la figura 2.1 (pág. 59). donde las implicaciones horizontales están dadas por las definiciones de los puntos fijos. El teorema muestra que si alguna de las implicaciones hacia abajo (en el diagrama) es válida las demás también lo son. Además el diagrama muestra que si P es un invariante entonces $\mu.\mathcal{F}_{\mathcal{T},\Theta}$ es un subconjunto de $\nu.\mathcal{B}_{\mathcal{T},P}$ como muestra la figura 2.2 (pág. 59).

Otra relación interesante es la que muestra la dualidad entre ambos puntos fijos:

Teorema 2.27 (Dualidad entre puntos fijos)
Dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ se cumple:

$$\mu.\mathcal{F}_{\mathcal{T},\Theta} = \neg\nu.\mathcal{B}_{\mathcal{T}^{-1},\neg\Theta} \ . \qquad \Box$$

Figura 2.1: Diagrama entre predicados cuando $\Box P$.

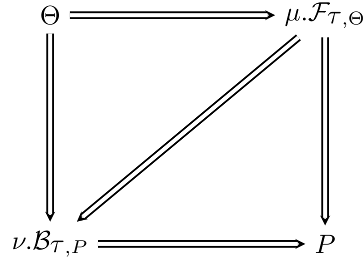
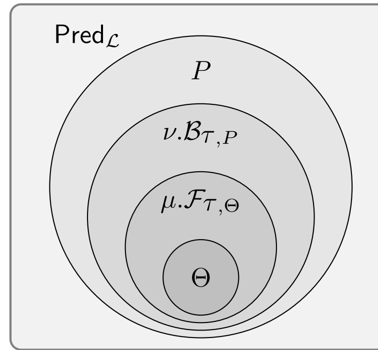


Figura 2.2: Relación entre los puntos fijos si $\Box P$.



Demostración

Del teorema 1.23 (pág. 27) se deduce la siguiente dualidad de los transformadores de propagación hacia adelante y atrás:

$$\mathcal{F}_{\mathcal{T},\Theta}.X = \neg\mathcal{B}_{\mathcal{T}^{-1},\neg\Theta}.\neg X \ .$$

Utilizando este resultado se demuestra el teorema:

$$
\begin{aligned}
& \mu.\mathcal{F}_{\mathcal{T},\Theta} \\
= & \ \{ \text{ Definición de } \mu.\mathcal{F}_{\mathcal{T},\Theta}\text{, pág. 48 } \} \\
& \langle \bigcap X \ : \mathcal{F}_{\mathcal{T},\Theta}.X \subseteq X : \ X \rangle \\
= & \ \{ \text{ Ley de Morgan } \} \\
& \neg \langle \bigcup X \ : \mathcal{F}_{\mathcal{T},\Theta}.X \subseteq X : \ \neg X \rangle \\
= & \ \{ \ \mathcal{F}_{\mathcal{T},\Theta}.X = \neg\mathcal{B}_{\mathcal{T}^{-1},\neg\Theta}.\neg X \ \} \\
& \neg \left\langle \bigcup X \ : \neg\mathcal{B}_{\mathcal{T}^{-1},\neg\Theta}.\neg X \subseteq X : \ \neg X \right\rangle \\
= & \ \{ \text{ Cambio de variable } X \text{ por } \neg Y \ \} \\
& \neg \left\langle \bigcup Y \ : \neg\mathcal{B}_{\mathcal{T}^{-1},\neg\Theta}.Y \subseteq \neg Y : \ Y \right\rangle \\
= & \ \{ \text{ Contrarecíproca de la implicación } \} \\
& \neg \left\langle \bigcup Y \ : Y \subseteq \mathcal{B}_{\mathcal{T}^{-1},\neg\Theta}.Y : \ Y \right\rangle
\end{aligned}
$$

$= \{$ Definición de $\nu\mathcal{B}_{\mathcal{T},P}$, pág. 53 $\}$

$\neg\nu.\mathcal{B}_{\mathcal{T}^{-1},\neg\Theta}$ □

2.3.4 Método de demostración de invariancia

En el ejemplo 2.25 vimos como la invariancia de un predicado puede ser demostrada calculando el punto fijo $\nu.\mathcal{B}_{\mathcal{T},P}$ si la secuencia $\{\mathcal{B}^i_{\mathcal{T},P}.\mathsf{True}\}_{i\geqslant 0}$ converge. A continuación mostraremos la validez del método desde otra perspectiva que lo relaciona al principio de inducción. Esto nos permitirá derivar su implementación además de explorar las limitaciones que posee.

La definición 2.10 conceptualiza la noción de invariante como un predicado que se cumple en todas las configuraciones durante la ejecución de un programa: dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ un predicado P es invariante cuando se verifica

$$\langle \forall\, \varrho, l, \sigma \ :\ \varrho \in [\![\mathsf{TS}]\!] \wedge (l,\sigma) \in \rho :\ P.l.\sigma \,\rangle \ .$$

Definamos de forma inductiva el conjunto de predicados $\mathcal{C} : \mathsf{Pred}_{\mathcal{L}} \to \mathsf{Bool}$ como la menor conjunto tal que

1. $\Theta \in \mathcal{C}$,

2. $X \in \mathcal{C}$ entonces SP $\mathcal{T}.X \in \mathcal{C}$.

Por lo visto en la sección 1.7 (pág. 28) el predicado SP.$\mathcal{T}.Q$ puede ser interpretado operacionalmente como el conjunto de configuraciones alcanzables en un paso de ejecución desde Q

$$\mathrm{SP}.\mathcal{T}.X = \big\{(l',\sigma') \mid \big\langle \exists\, l,\sigma \ :\ X.l.\sigma :\ (l,\sigma) \rightsquigarrow (l',\sigma') \big\rangle\big\} \ ,$$

por lo tanto para demostrar la invariancia de P como fue conceptualizada en la definición 2.10 se puede probar que el predicado contiene todos los conjuntos definidos en $\mathcal{C}$, esto es

$$\langle \forall\, X \ :\ X \in \mathcal{C} :\ X \subseteq P \,\rangle \ .$$

El conjunto $\mathcal{C}$ junto con la relación lineal $X\,R\,Y \doteq \mathrm{SP}.\mathcal{T}.X = Y$ es bien fundado por lo que se puede probar la propiedad anterior por inducción:

$\langle \forall\, X \ :\ X \in \mathcal{C} :\ X \subseteq P \,\rangle$

$\Leftarrow \{$ Inducción $\}$

$\Theta \subseteq P \wedge \langle \forall\, X \ :\ X \in \mathcal{C} :\ X \subseteq P \Rightarrow \mathrm{SP}.\mathcal{T}.X \subseteq P \,\rangle$

$\equiv \{$ Dualidad SP/WLP, propiedad 1.22 (pág. 27) $\}$

$\Theta \subseteq P \wedge \langle \forall\, X \ :\ X \in \mathcal{C} :\ X \subseteq P \Rightarrow X \subseteq \mathrm{WLP}.\mathcal{T}.P \,\rangle$

$\Leftarrow \{$ Monotonía $\}$

$\Theta \subseteq P \wedge P \subseteq \mathrm{WLP}.\mathcal{T}.P$

De esta forma, a partir del principio de inducción se puede deducir la siguiente regla para probar invariancia:

$$\Theta \subseteq P \wedge P \subseteq \text{WLP}.\mathcal{T}.P \Rightarrow \Box P \ .$$

Notar que esta regla impone justamente las condiciones para que P sea un invariante inductivo (definición 2.12) lo cual no siempre sucede (ver ejemplo 2.25). Además esta regla muestra cierta analogía con el principio de inducción matemática: para probar la invariancia de un predicado φ primero se debe probar el caso base $\Theta \subseteq \varphi$ con el cual φ se cumple inicialmente y el paso inductivo $\varphi \subseteq \text{WLP}.\mathcal{T}.\varphi$ indica que si el predicado se verifica en algún paso de ejecución, entonces se satisface en el siguiente. Por esto mismo llamamos a φ invariante inductivo.

A partir de este desarrollo deduciremos una regla nueva para probar invariancia en base a otro principio de inducción denominado *inducción k* o *k-inducción*. En el mismo se fortalece el caso base y se debilita el paso inductivo: dado un natural k fijo, si quiero probar la validez de cierta propiedad $P.i$ para todo natural i primero se prueba $P.0 \wedge \cdots \wedge P.(k-1)$ (casos bases) y, suponiendo $P.j \wedge \cdots \wedge P.(j+k-1)$ se demuestra $P.(j+k)$. Formalmente:

$$
\begin{array}{ll}
 & \langle \forall i \ : 0 \leqslant i : \ P.i \rangle \\
\Leftarrow & \{ \ k\text{-inducción} \ \} \\
 & \langle \forall i \ : 0 \leqslant i < k : \ P.i \rangle \\
 & \ \wedge \langle \forall i \ : 0 \leqslant i : \langle \forall j \ : i \leqslant j < i+k : \ P.j \rangle \Rightarrow P.(i+k) \rangle
\end{array}
$$

Notar que cuando $k = 1$ es el principio de inducción estándar.

De igual manera, este principio de inducción puede utilizarse para obtener una regla de demostración de invariancia:

$$
\begin{array}{ll}
 & \langle \forall X \ : X \in \mathcal{C} : \ X \subseteq P \rangle \\
\Leftarrow & \{ \ k\text{-inducción} \ \} \\
 & \langle \forall i \ : 0 \leqslant i < k : \ (\text{SP}.\mathcal{T})^i.\Theta \subseteq P \rangle \\
 & \ \wedge \langle \forall X \ : X \in \mathcal{C} \ : \\
 & \qquad \langle \forall i \ : 0 \leqslant i < k : \ (\text{SP}.\mathcal{T})^i.X \subseteq P \rangle \Rightarrow (\text{SP}.\mathcal{T})^k.X \subseteq P \rangle \\
\equiv & \{ \ \text{Dualidad SP WLP, propiedad 1.22 (pág. 27)} \ \} \\
 & \langle \forall i \ : 0 \leqslant i < k : \ \Theta \subseteq (\text{WLP}.\mathcal{T})^i.P \rangle \\
 & \ \wedge \langle \forall X \ : X \in \mathcal{C} \ : \\
 & \qquad \langle \forall i \ : 0 \leqslant i < k : \ X \subseteq (\text{WLP}.\mathcal{T})^i.P \rangle \Rightarrow X \subseteq (\text{WLP}.\mathcal{T})^k.P \rangle \\
\equiv & \{ \ \text{Cálculo de predicados} \ \} \\
 & \Theta \subseteq \langle \bigcap i \ : 0 \leqslant i < k : \ (\text{WLP}.\mathcal{T})^i.P \rangle \\
 & \ \wedge \langle \forall X \ : X \in \mathcal{C} \ : \\
 & \qquad X \subseteq \langle \bigcap i \ : 0 \leqslant i < k : \ (\text{WLP}.\mathcal{T})^i.P \rangle \Rightarrow X \subseteq (\text{WLP}.\mathcal{T})^k.P \rangle \\
\Leftarrow & \{ \ \text{Monotonía} \subseteq \ \} \\
 & \Theta \subseteq \langle \bigcap i \ : 0 \leqslant i < k : \ (\text{WLP}.\mathcal{T})^i.P \rangle \\
 & \ \wedge \langle \bigcap i \ : 0 \leqslant i < k : \ (\text{WLP}.\mathcal{T})^i.P \rangle \subseteq (\text{WLP}.\mathcal{T})^k.P
\end{array}
$$

$\equiv$ { Monotonía $\subseteq$ }

$\Theta \subseteq \langle \bigcap i \ : 0 \leqslant i < k : \ (\text{WLP}.\mathcal{T})^i.P \rangle$

$\wedge \langle \bigcap i \ : 0 \leqslant i < k : \ (\text{WLP}.\mathcal{T})^i.P \rangle$

$\quad \subseteq \langle \bigcap i \ : 0 \leqslant i < k+1 : \ (\text{WLP}.\mathcal{T})^i.P \rangle$

$\equiv$ { Caracterización de $\mathcal{B}^n_{\mathcal{T},P}.\mathsf{True}$ (pág. 55) }

$\Theta \subseteq \mathcal{B}^k_{\mathcal{T},P}.\mathsf{True}$

$\wedge\ \mathcal{B}^k_{\mathcal{T},P}.\mathsf{True} \subseteq \mathcal{B}^{k+1}_{\mathcal{T},P}.\mathsf{True}$

De esta forma, a partir del principio de k-inducción se puede deducir la siguiente regla para probar invariancia:

$$\Theta \subseteq \mathcal{B}^k_{\mathcal{T},P}.\mathsf{True} \wedge \mathcal{B}^k_{\mathcal{T},P}.\mathsf{True} \subseteq \mathcal{B}^{k+1}_{\mathcal{T},P}.\mathsf{True} \Rightarrow \Box P \ .$$

Notar que la condición $\mathcal{B}^k_{\mathcal{T},P}.\mathsf{True} \subseteq \mathcal{B}^{k+1}_{\mathcal{T},P}.\mathsf{True}$ indica que la secuencia $\{\mathcal{B}^i_{\mathcal{T},P}.\mathsf{True}\}_{i\geqslant 0}$ converge en k pasos a $\mathcal{B}^k_{\mathcal{T},P}.\mathsf{True}$. Además esta condición es equivalente a $\mathcal{B}^k_{\mathcal{T},P}.\mathsf{True} \subseteq \text{WLP}.\mathcal{T}.(\mathcal{B}^k_{\mathcal{T},P}.\mathsf{True}) \wedge \mathcal{B}^k_{\mathcal{T},P}.\mathsf{True} \subseteq P$ y junto con la condición $\Theta \subseteq \mathcal{B}^k_{\mathcal{T},P}.\mathsf{True}$ se deduce que $\mathcal{B}^k_{\mathcal{T},P}.\mathsf{True}$ es un invariante inductivo más fuerte que P. Si esto sucede diremos que P es un k-*invariante* . Es fácil de ver que si P es k-invariante entonces también es m-invariante para todo $m \geqslant k$.

A partir de este resultado podemos derivar el siguiente método de demostración de invariancia: para k desde 0 intentamos verificar si P es un k-invariante. Si la condición $\Theta \subseteq \mathcal{B}^k_{\mathcal{T},P}.\mathsf{True}$ (caso base k-inductivo) no es verdadera se puede concluir que P no es invariante ya que entonces tampoco lo será $\Theta \subseteq \nu.\mathcal{B}_{\mathcal{T},P}$ (recordar que $\nu.\mathcal{B}_{\mathcal{T},P} \subseteq \mathcal{B}^k_{\mathcal{T},P}.\mathsf{True}$ ya que es límite de la cadena descendente) y no se cumplirá la condición del teorema 2.24. Si esta condición se cumple entonces se procede a verificar $\mathcal{B}^k_{\mathcal{T},P}.\mathsf{True} \subseteq \mathcal{B}^{k+1}_{\mathcal{T},P}.\mathsf{True}$ (paso k-inductivo). Si esta proposición es verdadera el método termina concluyendo que P es un k-invariante. Si no se cumple se procede a intentar demostrar si P es un $(k+1)$-invariante inductivo. El método se puede esquematizar con el programa 2.3.

Programa 2.3 Demostración de k-invariancia

es_invariante ($P, \Theta : \mathsf{Pred}_\mathcal{L}$; $\mathcal{T} : \mathsf{Tran}_{\mathcal{L},\mathcal{S}}$) : Bool

$\quad \mathcal{B}_{\mathcal{T},P} := \langle \lambda X \bullet P \cap \text{WLP}.\mathcal{T}.X \rangle$;

$\quad B_k := \mathsf{True}$;

$\quad$ **do** $\Theta \subseteq B_k \wedge \neg(B_k \subseteq \mathcal{B}_{\mathcal{T},P}.B_k)$

$\qquad B_k := \mathcal{B}_{\mathcal{T},P}.B_k$

$\quad$ **od**;

$\quad$ **return** $\Theta \subseteq B_k$

Una pregunta importante es si el método termina para cualquier sistema de transiciones e invariante candidato P. Claramente, esto es equivalente a preguntarnos si existe algún k tal que por k-inducción se puede demostrar la invariancia. Suponiendo que poseemos métodos de decisión completos para resolver las relaciones de orden entre los predicados de la guarda del bucle, las causas

de terminación son la no verificación de la condición $\Theta \subseteq \mathcal{B}^k_{\mathcal{T},P}.\mathsf{True}$ en el caso que P no sea invariante, o la verificación de $\mathcal{B}^k_{\mathcal{T},P}.\mathsf{True} \subseteq \mathcal{B}^{k+1}_{\mathcal{T},P}.\mathsf{True}$ para el caso que lo sea. Analicemos el primer caso: supongamos que P no es invariante y se cumple $\langle \forall\, k\ : 0 \leqslant k :\ \Theta \subseteq \mathcal{B}^k_{\mathcal{T},P}.\mathsf{True} \rangle$.

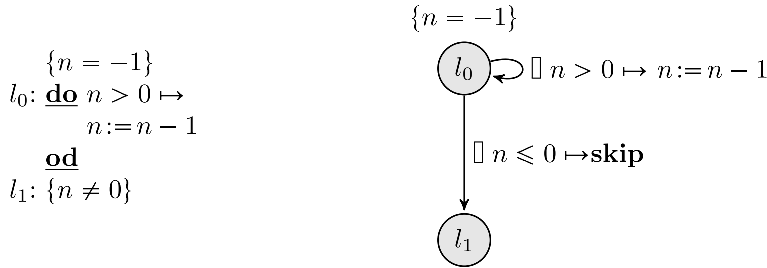
Comenzando de esta última suposición obtenemos la siguiente equivalencia.

$$\langle \forall\, k\ : 0 \leqslant k :\ \Theta \subseteq \mathcal{B}^k_{\mathcal{T},P}.\mathsf{True} \rangle$$
$$\equiv \{ \text{ Cálculo de predicados } \}$$
$$\Theta \subseteq \langle \bigcap\, k\ : 0 \leqslant k :\ \mathcal{B}^k_{\mathcal{T},P}.\mathsf{True} \rangle$$
$$\equiv \{ \text{ Caracterización de } \nu.\mathcal{B}_{\mathcal{T},P} \}$$
$$\Theta \subseteq \nu.\mathcal{B}_{\mathcal{T},P}$$
$$\equiv \{ \text{ Teorema 2.24 } \}$$
$$\Box P$$

Con lo cual hemos llegado a una contradicción. A partir de este razonamiento por el absurdo se puede concluir que existe k tal que no se cumple $\Theta \subseteq \mathcal{B}^k_{\mathcal{T},P}.\mathsf{True}$ y esto quiere decir que en el caso que P no sea invariante el método termina.

La otra posibilidad es que el método no termine debido a que nunca se cumple la condición $\mathcal{B}^k_{\mathcal{T},P}.\mathsf{True} \subseteq \mathcal{B}^{k+1}_{\mathcal{T},P}.\mathsf{True}$ para todo k ya que la cadena descendente $\{\mathcal{B}^i_{\mathcal{T},P}.\mathsf{True}\}_{i \geqslant 0}$ es infinita. Lamentablemente esto puede suceder aún para sistemas de transiciones muy simples. El programa 2.4 es un ejemplo de este fenómeno.

Programa 2.4 Contraejemplo de terminación



Sobre el mismo se puede calcular $\mathcal{B}^k_{\mathcal{T},P}.\mathsf{True}$:

$$\begin{array}{lcl}
\mathcal{B}_{\mathcal{T},P}.\mathsf{True} & = & [\mathsf{True},\ n \neq 0] \\
\mathcal{B}^2_{\mathcal{T},P}.\mathsf{True} & = & [n \neq 0,\ n \neq 0] \\
\mathcal{B}^3_{\mathcal{T},P}.\mathsf{True} & = & [n \neq 0 \wedge n \neq 1,\ n \neq 0] \\
\mathcal{B}^4_{\mathcal{T},P}.\mathsf{True} & = & [n \neq 0 \wedge n \neq 1 \wedge n \neq 2,\ n \neq 0] \\
\vdots & &
\end{array}$$

Se puede demostrar por inducción que $\mathcal{B}^{k+1}_{\mathcal{T},P}.\mathsf{True} = [n < 0\ \vee\ n \geqslant k,\ n \neq 0]$. Todos los elementos de esta secuencia son implicados por $\Theta\ \doteq\ [n = -1,\ \mathsf{False}]$ pero la secuencia no converge en un número finito de pasos y su ínfimo es $[n < 0,\ n \neq 0]$ ($n \neq 0$ al terminar es invariante).

En el desarrollo de esta sección nos hemos enfocado en el método para probar invariancia basado en el transformador de propagación hacia atrás. Como se mostró en el ejemplo 2.20 también es posible utilizar el transformador de propagación hacia adelante para calcular el mínimo punto fijo $\mu.\mathcal{F}_{\mathcal{T},\Theta}$. El método es esquematizado en el programa 2.5.

Programa 2.5 Método utilizando propagación hacia adelante

es_invariante' (P, Θ : $\mathsf{Pred}_{\mathcal{L}}$; $\mathcal{T}$: $\mathsf{Tran}_{\mathcal{L},\mathcal{S}}$) : Bool

$\mathcal{F}_{\mathcal{T},\Theta} := \langle \lambda X \bullet \Theta \cup \mathrm{SP}.\mathcal{T}.X \rangle$;

$F_k := \mathsf{False}$;

do $F_k \subseteq P \wedge \neg(\mathcal{F}_{\mathcal{T},\Theta}.F_k \subseteq F_k)$

$F_k := \mathcal{F}_{\mathcal{T},\Theta}.F_k$

od;

return $F_k \subseteq P$

Como veremos el método puede ser implementado también con el programa 2.3 en la forma **es_invariante**$(\neg\Theta, \neg P, \mathcal{T}^{-1})$: reemplazando P, Θ y $\mathcal{T}$ por $\neg\Theta$, $\neg P$ y $\mathcal{T}^{-1}$ en aquel programa, el cálculo de $\mathcal{B}_{\mathcal{T}^{-1},\neg\Theta}.X$ es equivalente a calcular $\neg\mathcal{F}_{\mathcal{T},\Theta}.\neg X$ (por teorema 2.27), por lo que la variable B_k almacenará $\neg F_k$. De esta manera el programa anterior calculará la misma cadena negada. Además, con estos remplazos la guarda del bucle es $(\neg P) \subseteq B_k \wedge \neg(B_k \subseteq \mathcal{B}_{\mathcal{T}^{-1},\neg\Theta}.B_k)$ lo cual, mediante el razonamiento anterior, es equivalente a $F_k \subseteq P \wedge \neg(\mathcal{F}_{\mathcal{T},\Theta}.F_k \subseteq F_k)$. De tal forma, la dualidad de los métodos es corroborada por el anterior teorema 2.27 obteniendose la siguiente equivalencia:

$$\textbf{es_invariante}'(P,\Theta,\mathcal{T}) \equiv \textbf{es_invariante}(\neg\Theta,\neg P,\mathcal{T}^{-1}) \ .$$

El último método tiene la ventaja de poder obtener el invariante más fuerte del sistema de transiciones independientemente de la propiedad P que se quiera verificar: si ya obtuvimos el invariante $\mu.\mathcal{F}_{\mathcal{T},\Theta}$, este puede reutilizarse para probar la invariancia de cualquier propiedad P' verificando la validez de la implicación $\nu.\mathcal{F}_{\mathcal{T},\Theta} \subseteq P'$. Aunque el método parece promisorio, su principal desventaja es que el cálculo del transformador SP involucra la aparición de cuantificadores existenciales al ser aplicado a asignaciones guardadas no invertibles[1], como se muestra en el ejemplo 1.3 (pág. 11). Esto complica la demostración automática de las implicaciones que aparecen en el método como se explica en [TRSS01]. Cuando se utiliza la propagación hacia atrás este fenómeno no aparece debido a la eliminación que se produce del cuantificador universal al ser aplicado sobre este tipo de sentencias (como vimos en el ejemplo 1.5, pág. 12).

Otra debilidad del método es que suele no converger en casos donde el método de propagación hacia atrás si lo hace. Esto se debe a que por lo general los programas no alcanzan todos los estados posibles en una cantidad finita de iteraciones. El fenómeno es mostrado en los ejemplos 2.20 y 2.25 y además es confirmado en [BBM97]. Por lo tanto para su implementación efectiva se

[1]Como vimos en la sección 1.5.2 (pág. 21) estas sentencias caracterizan los sistemas de transiciones ejecutables.

hace necesaria la aplicación de técnicas de interpretación abstracta que serán explicadas en el capítulo siguiente. Por estas razones nuestro trabajo emplea principalmente el primer método.

Capítulo 3

Generación de invariantes lineales

Como vimos en el capítulo anterior la secuencia $\{\mathcal{F}^i_{\mathcal{T},\Theta}.\mathsf{False}\}_{i \geqslant 0}$ suele ser infinita por lo que el método esquematizado en el programa 2.5 puede no terminar. Una forma posible de atacar este problema es por medio de la teoría de *Interpretación Abstracta* [CC77, CC92]. A grandes rasgos esta base teórica presenta distintos marcos de trabajo con los cuales se aproxima la semántica de strongest postcondition presentada en la sección 1.6 (pág. 22) mediante transformadores sobre otros dominios distintos a $\mathsf{Pred}_{\mathcal{L}}$. De esta manera se lleva el problema a dominios donde existen procedimientos que permiten obtener propiedades de forma automática. A continuación explicaremos el marco de trabajo utilizado dentro de la teoría general de interpretación abstracta. Un resumen y análisis de todos los marcos de interpretación abstracta puede encontrarse en [CC92].

3.1 Interpretación Abstracta

De forma general, la interpretación abstracta puede definirse como una serie de métodos desarrollados a partir de distintos marcos de trabajo para diseñar aproximaciones de la semántica. Los mismos pueden ser utilizados para extraer información de los programas con la finalidad de obtener algunas de sus propiedades de forma automática. Debido al objetivo de mecanización en la extracción de propiedades, las respuestas suelen ser aproximadas ya que el problema general es indecidible, con lo cual los métodos desarrollados aunque correctos suelen ser incompletos.

El marco de trabajo de interpretación abstracta utilizado en esta tesis consiste en la identificación de cuatro entidades del problema: dominio semántico concreto, dominio semántico abstracto, función de concretización y operador de widening.

3.1.1 Dominio semántico concreto

Dentro del marco general de interpretación abstracta, un *dominio semántico concreto* se identifica a partir de una abstracción de la *semántica estándar* de los programas que nos permita focalizarnos en las propiedades que nos interesan.

Generalmente la semántica estándar esta dada por el conjunto de trazas de un sistema (*semántica operacional*), en el sentido desarrollado en la sección 1.7 (pág. 28), o por su semántica denotacional. Nos concentraremos en el primer punto de vista ya que dentro del mismo se desarrolló nuestro trabajo[1].

Ejemplo 3.1
Como vimos en la sección 2.3.1 (pág. 48) sobre $\mathsf{Pred}_{\mathcal{L}}$ se pueden caracterizar propiedades de invariancia acordes a la semántica operacional (*semántica de trazas*) a través del punto fijo del transformador de propagación hacia adelante. En este caso nuestra semántica estándar es la semántica operacional y el dominio semántico concreto será $\mathsf{Pred}_{\mathcal{L}}$.

Notar además que la semántica generada por el transformador resulta una abstracción de la semántica de trazas ya que nos concentramos solo en las propiedades de invariancia de los sistemas, dejando de lado otras propiedades temporales. □

Teniendo este ejemplo en mente, el marco de interpretación abstracta supone la existencia de una *función semántica concreta* $f : \mathcal{P} \to \mathcal{P}$ donde $\mathcal{P}$ es el dominio semántico concreto y cuyo mínimo punto fijo describe las propiedades de invariancia del sistema. Para la existencia del punto fijo es necesario que el dominio semántico sea una de las estructuras señaladas en la sección 2.1 (pág. 39).

Ejemplo 3.2
Como ya mencionamos, un dominio semántico concreto puede ser $\mathsf{Pred}_{\mathcal{L}}$ y la función semántica concreta será $\mathcal{F}_{\mathcal{T},\Theta} : \mathsf{Pred}_{\mathcal{L}} \to \mathsf{Pred}_{\mathcal{L}}$. En este caso el dominio es un cpo y la función semántica es $\sqcup$-continua lo que garantiza la existencia del mínimo punto fijo. □

3.1.2 Dominio semántico abstracto y abstracción

El siguiente paso consiste en definir un poset $(\mathcal{P}^\sharp, \sqsubseteq^\sharp)$ donde pueda aproximarse el invariante obtenido por el mínimo punto fijo de la función semántica concreta. El poset será denominado *dominio semántico abstracto* y deberá tener mínimo elemento $\bot^\sharp$. A partir de aquí usaremos la notación $\Gamma^\sharp \doteq (\mathcal{P}^\sharp, \sqsubseteq^\sharp)$ para denotar el dominio abstracto y $\Gamma \doteq (\mathcal{P}, \sqsubseteq)$ para el concreto

Dado el dominio semántico concreto $(\mathcal{P}, \sqsubseteq)$, a partir del dominio abstracto $(\mathcal{P}^\sharp, \sqsubseteq^\sharp)$ se procederá a definir una función monótona e inyectiva $\gamma : \mathcal{P}^\sharp \to \mathcal{P}$ denominada *función de concretización* la cual permite derivar una noción de corrección del sistema en el dominio abstracto, esto es

. sea un programa con $f : \mathcal{P} \to \mathcal{P}$ su función semántica concreta y

. $x^\sharp \in \mathcal{P}^\sharp$ con $\mu.f \sqsubseteq \gamma.x^\sharp$ ($\mu.f$ es el mínimo punto fijo de f)

entonces $x^\sharp$ expresa una propiedad abstracta válida del programa.

Notar además que por ser γ monótona, si un elemento del dominio abstracto es válido para un sistema entonces cualquier elemento mayor (en el orden $\sqsubseteq^\sharp$) también lo es.

[1] En [CC92] puede encontrarse un ejemplo de utilización de la semántica denotacional como semántica estándar.

La función de concretización será la base con la cual se define la aproximación de los elementos del dominio concreto. En este sentido, cuando $x \in \mathcal{P}$ y $x^\sharp \in \mathcal{P}^\sharp$ cumplan $x \sqsubseteq \gamma.x^\sharp$ diremos que $x^\sharp$ es una *abstracción de x*. Además la terna $(\Gamma, \Gamma^\sharp, \gamma)$ será llamada simplemente *abstracción*.

3.1.3 Función semántica abstracta

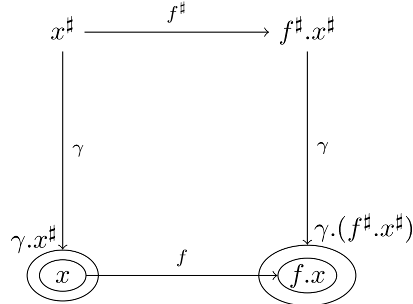
Una vez definida la función de concretización se procede a aproximar la función semántica concreta. Para ello utilizaremos el siguiente concepto.

Definición 3.3 (Función semántica abstracta)
Dada una abstracción $(\Gamma, \Gamma^\sharp, \gamma)$ con la función semántica concreta $f : \mathcal{P} \to \mathcal{P}$, una función monótona $f^\sharp : \mathcal{P}^\sharp \to \mathcal{P}^\sharp$ será llamada *función semántica abstracta* de f si

$$\left\langle \forall\, x, x^\sharp\ : x \in \mathcal{P} \wedge x^\sharp \in \mathcal{P}^\sharp :\ x \sqsubseteq \gamma.x^\sharp \Rightarrow f.x \sqsubseteq \gamma.(f^\sharp.x^\sharp) \right\rangle \ . \qquad \square$$

Esta definición formaliza la noción de aproximación de la función semántica concreta f por la función $f^\sharp$. En el diagrama 3.1 se esquematiza esta relación.

Figura 3.1: Propiedad de la función semántica abstracta



La ultima definición es frecuentemente utilizada en la literatura [CC92, Mon00]. El siguiente lema muestra una caracterización más simple de la función semántica abstracta.

Lema 3.4 (Función semántica abstracta)
Dada una abstracción $(\Gamma, \Gamma^\sharp, \gamma)$ con la función semántica concreta $f : \mathcal{P} \to \mathcal{P}$. Una función $f^\sharp : \mathcal{P}^\sharp \to \mathcal{P}^\sharp$ es una *función semántica abstracta* de f si y solo si para todo $x^\sharp \in \mathcal{P}^\sharp$ se cumple $f.(\gamma.x^\sharp) \sqsubseteq \gamma.(f^\sharp.x^\sharp)$. $\square$

Demostración

($\Rightarrow$) Por reflexividad del poset $(\mathcal{P}^\sharp, \sqsubseteq)$ se cumple $\gamma.x^\sharp \sqsubseteq \gamma.x^\sharp$, entonces, utilizando la definición anterior, $f.(\gamma.x^\sharp) \sqsubseteq \gamma.(f^\sharp.x^\sharp)$ por ser $f^\sharp$ una función semántica abstracta.

($\Leftarrow$) Supongamos $x \in \mathcal{P}$ y $x^\sharp \in \mathcal{P}^\sharp$ tal que $x \sqsubseteq \gamma.x^\sharp$, entonces

$$
\begin{array}{ll}
 & f.x \\
\sqsubseteq & \{ \text{ Suposición y } f \text{ monótona } \} \\
 & f.(\gamma.x^\sharp) \\
\sqsubseteq & \{ \text{ Hipótesis de caso de prueba } \} \\
 & \gamma.(f^\sharp.x^\sharp)
\end{array}
$$

Por transitividad $f.x \sqsubseteq \gamma.(f^\sharp.x^\sharp)$ y por lo tanto $f^\sharp$ es una función semántica abstracta de f. □

La idea general de este marco de interpretación abstracta es aproximar el punto fijo $\mu.f$ de la función semántica concreta calculando el límite de la función $f^\sharp$ en el dominio abstracto. Para ello utilizaremos el siguiente teorema.

Teorema 3.5
Dada una abstracción $(\Gamma, \Gamma^\sharp, \gamma)$ con $\Gamma \doteq (\mathcal{P}, \sqsubseteq)$ un cpo, $\Gamma^\sharp \doteq (\mathcal{P}^\sharp, \sqsubseteq^\sharp)$ un poset con mínimo elemento $\perp^\sharp$, $f : \mathcal{P} \to \mathcal{P}$ una función semántica concreta $\sqcup$-continua y $f^\sharp$ una función semántica abstracta monótona, entonces:

1. el supremo $\left\langle \bigsqcup i \ : i \in \mathbb{N} : \ \gamma.(f^{\sharp(i)}.\perp^\sharp) \right\rangle$ existe, y
2. es mayor al mínimo punto fijo de f, esto es

$$\mu.f \sqsubseteq \left\langle \bigsqcup i \ : i \in \mathbb{N} : \ \gamma.(f^{\sharp(i)}.\perp^\sharp) \right\rangle \ . \qquad \square$$

Demostración
1. Por monotonía de $f^\sharp$ y γ, el conjunto $\left\{\gamma.(f^{\sharp(i)}.\perp^\sharp) \mid i \in \mathbb{N}\right\}$ es dirigido (es una cadena ascendente) por lo tanto como Γ es un cpo, el supremo existe.
2. Primero demostremos por inducción que $f^i.\perp \sqsubseteq \gamma.(f^{\sharp(i)}.\perp^\sharp)$ para todo $i \in \mathbb{N}$. El caso base se demuestra fácilmente ya que $\perp \sqsubseteq \gamma.\perp^\sharp$ por ser el mínimo elemento de Γ.

   Para el paso inductivo supondremos $f^i.\perp \sqsubseteq \gamma.(f^{\sharp(i)}.\perp^\sharp)$. Por definición 3.3 entonces se cumple $f^{(i+1)}.\perp \sqsubseteq \gamma.(f^{\sharp(i+1)}.\perp^\sharp)$. Con esta demostración concluimos $\left\langle \forall i \ : i \in \mathbb{N} : \ f^i.\perp \sqsubseteq \gamma.(f^{\sharp(i)}.\perp^\sharp) \right\rangle$ con lo cual:

$$
\begin{array}{ll}
 & \left\langle \forall i \ : i \in \mathbb{N} : \ f^i.\perp \sqsubseteq \gamma.(f^{\sharp(i)}.\perp^\sharp) \right\rangle \\
\Rightarrow & \{ \text{ Existencia y definición de supremos en conjuntos dirigidos } \} \\
 & \left\langle \forall i \ : i \in \mathbb{N} : \ f^i.\perp \sqsubseteq \left\langle \bigsqcup j \ : j \in \mathbb{N} : \ \gamma.(f^{\sharp(j)}.\perp^\sharp) \right\rangle \right\rangle \\
\equiv & \{ \text{ Definición de supremo } \} \\
 & \left\langle \bigsqcup i \ : i \in \mathbb{N} : \ f^i.\perp \right\rangle \sqsubseteq \left\langle \bigsqcup j \ : j \in \mathbb{N} : \ \gamma.(f^{\sharp(j)}.\perp^\sharp) \right\rangle \\
\equiv & \{ \text{ Punto fijo de funciones continuas (teorema 2.8) } \} \\
 & \mu.f \sqsubseteq \left\langle \bigsqcup j \ : j \in \mathbb{N} : \ \gamma.(f^{\sharp(j)}.\perp^\sharp) \right\rangle
\end{array}
$$
□

Este teorema nos brinda un método para aproximar en el dominio abstracto el mínimo punto fijo $\mu.f$ similar al descripto en el programa 2.5 (pág. 64): dada una función semántica abstracta $f^\sharp : \mathcal{P}^\sharp \to \mathcal{P}^\sharp$ de la concreta f, se comienza el proceso iterativo desde el mínimo abstracto $\perp^\sharp$ y se calcula la cadena $\{\perp^\sharp, f^\sharp.\perp^\sharp, f^{\sharp(2)}.\perp^\sharp, \cdots\}$ hasta que $\gamma.(f^{\sharp(k)}.\perp^\sharp)$ sea igual a $\gamma.(f^{\sharp(k+1)}.\perp^\sharp)$.

Notar además que como la función de concretización γ es inyectiva y monótona esta última verificación puede realizarse sobre el dominio abstracto como $f^{\sharp(k+1)}.\bot^\sharp \sqsubseteq^\sharp f^{\sharp(k)}.\bot^\sharp$. Con esta última modificación el método puede esquematizarse con el programa 3.1.

Programa 3.1 Aproximación de $\mu.f$

aproximar_$\mu.f$ $(f^\sharp : \mathcal{P}^\sharp \to \mathcal{P}^\sharp) : \mathcal{P}^\sharp$
$F_k^\sharp := \bot^\sharp$;
do $\neg(f^\sharp.F_k^\sharp \sqsubseteq^\sharp F_k^\sharp)$
$F_k^\sharp := f^\sharp.F_k^\sharp$
od;
return $F_k^\sharp$

Este método presenta dos problemas: el procedimiento intenta calcular efectivamente el supremo $\left\langle \bigsqcup i \ : i \in \mathbb{N} : \ f^{\sharp(i)}.\bot^\sharp \right\rangle$ el cual puede no existir ya que $\mathcal{P}^\sharp$ no es necesariamente un cpo. Además, aunque exista, el procedimiento puede no terminar debido a que la cadena $\{\bot^\sharp, f^\sharp.\bot^\sharp, f^{\sharp(2)}.\bot^\sharp, \cdots\}$ puede ser infinita. Ambos inconvenientes son atacados en la teoría de interpretación abstracta como veremos en la sección siguiente.

3.1.4 Widening

Con el fin de asegurar la convergencia del método esquematizado en el programa 3.1 este marco de interpretación abstracta postula una manera de transformar la secuencia $\{\bot^\sharp, f^\sharp.\bot^\sharp, f^{\sharp(2)}.\bot^\sharp, \cdots\}$ mediante la aplicación de un operador definido en el dominio semántico abstracto.

Definición 3.6 (Operador de widenig)
Dado un dominio semántico abstracto $\Gamma^\sharp \doteq (\mathcal{P}^\sharp, \sqsubseteq^\sharp)$, un operador binario $_ \nabla _ : \mathcal{P}^\sharp \to \mathcal{P}^\sharp \to \mathcal{P}^\sharp$ será llamado *operador de widening* si

1. para todo $x, y : \mathcal{P}^\sharp$ se cumple

$$x \sqsubseteq^\sharp x \nabla y \quad \text{y} \quad y \sqsubseteq^\sharp x \nabla y$$

2. para toda cadena ascendente $x_0 \sqsubseteq^\sharp x_1 \sqsubseteq^\sharp x_2 \sqsubseteq^\sharp \cdots$ en $\mathcal{P}^\sharp$ la cadena $x'_0, x'_1, x'_2, \cdots$ definida como $x'_0 \doteq x_0$ y $x'_{i+1} \doteq x'_i \nabla x_{i+1}$ converge en un número finito de pasos. □

Notar que así definida, $x'_0, x'_1, x'_2, \cdots$ es una cadena ascendente ya que $x'_i \sqsubseteq^\sharp x'_{i+1}$ por el punto anterior.

Teorema 3.7
Dada una abstracción $(\Gamma, \Gamma^\sharp, \gamma)$ con $\Gamma \doteq (\mathcal{P}, \sqsubseteq)$ un cpo, $\Gamma^\sharp \doteq (\mathcal{P}^\sharp, \sqsubseteq^\sharp)$ un poset con mínimo elemento $\bot^\sharp$, $f : \mathcal{P} \to \mathcal{P}$ una función semántica concreta $\sqcup$-continua, $f^\sharp : \mathcal{P}^\sharp \to \mathcal{P}^\sharp$ una función semántica abstracta monótona y $_ \nabla _ : \mathcal{P}^\sharp \to \mathcal{P}^\sharp \to \mathcal{P}^\sharp$ un operador de widening.

Sea además, la función g definida de forma recursiva:

$$\begin{array}{|l} g^\sharp : \mathbb{N} \to \mathcal{P}^\sharp \\ \hline g^\sharp.0 \ \doteq \ \bot^\sharp \\ g^\sharp.(i+1) \ \doteq \ g^\sharp.i \ \nabla \ f^\sharp.(g^\sharp.i) \end{array}$$

entonces:

1. Los elementos del conjunto $\{g^\sharp.i \mid i \in \mathbb{N}\}$ forman una cadena ascendente finita.

2. La concretización del máximo de este conjunto es mayor o igual al mínimo punto fijo de f, esto es

$$\mu.f \sqsubseteq \gamma.\left\langle \bigsqcup^\sharp i \ : i \in \mathbb{N} : \ g^\sharp.i \right\rangle \ .$$

□

Demostración

1. Veamos primero que el conjunto es una cadena ascendente:

$$\begin{array}{ll} & g^\sharp.i \sqsubseteq^\sharp g^\sharp.(i+1) \\ \equiv & \{ \text{ Definición de } g^\sharp \text{ } \} \\ & g^\sharp.i \sqsubseteq^\sharp g^\sharp.i \ \nabla \ f^\sharp.(g^\sharp.i) \\ \equiv & \{ \text{ Definición 3.6 de operador de widening } \} \\ & \mathsf{T} \end{array}$$

En consecuencia $g^\sharp.i \sqsubseteq^\sharp g^\sharp.(i+1)$, para todo $i \in \mathbb{N}$, y como $f^\sharp$ es monótona también se cumple $f^\sharp.(g^\sharp.i) \sqsubseteq^\sharp f^\sharp.(g^\sharp.(i+1))$. Por lo tanto

$$f^\sharp.(g^\sharp.0) \sqsubseteq^\sharp f^\sharp.(g^\sharp.1) \sqsubseteq^\sharp f^\sharp.(g^\sharp.2) \sqsubseteq^\sharp \cdots$$

es una cadena ascendente. En la definición de widening 3.6, reemplazando x_i por $f^\sharp.(g^\sharp.i)$ y x'_i por $g^\sharp.i$ se deduce que $g^\sharp.0 \sqsubseteq^\sharp g^\sharp.1 \sqsubseteq^\sharp g^\sharp.2 \sqsubseteq^\sharp \cdots$ es una cadena finita.

2. Veamos primero por inducción que $f^{\sharp(i)}.\bot^\sharp \sqsubseteq^\sharp g^\sharp.i$.

Caso base

$$\begin{array}{ll} & f^{\sharp(0)}.\bot^\sharp \sqsubseteq^\sharp g^\sharp.0 \\ \equiv & \{ \text{ Definición de } f^\sharp \text{ y } g^\sharp \text{ } \} \\ & \bot^\sharp \sqsubseteq^\sharp \bot^\sharp \\ \Leftarrow & \{ \text{ Reflexividad de } \sqsubseteq^\sharp \text{ } \} \\ & \mathsf{T} \end{array}$$

Paso inductivo (Hipótesis inductiva $f^{\sharp(i)}.\bot^\sharp \sqsubseteq^\sharp g^\sharp.i$)

$$f^{\sharp(i+1)}.\bot^\sharp$$

$\sqsubseteq^\sharp$ { Hip. Inductiva y $f^\sharp$ monótona }
$f^\sharp.(g^\sharp.i)$
$\sqsubseteq^\sharp$ { Definición 3.6 de operador de widening }
$g^\sharp.i \ \nabla \ f^\sharp.(g^\sharp.i)$
$=$ { Definición de $g^\sharp$ }
$g^\sharp.(i+1)$

En consecuencia, para todo $i \in \mathbb{N}$ se cumple $f^{\sharp(i)}.\bot^\sharp \sqsubseteq^\sharp g^\sharp.i$. A partir de este resultado se demuestra el segundo ítem del teorema:

$\left\langle \forall i \ : i \in \mathbb{N} : \ f^{\sharp(i)}.\bot^\sharp \sqsubseteq^\sharp g^\sharp.i \right\rangle$
$\Rightarrow$ { $\{g^\sharp.i \mid i \in \mathbb{N}\}$ es una cadena finita }
$\left\langle \forall i \ : i \in \mathbb{N} : \ f^{\sharp(i)}.\bot^\sharp \sqsubseteq^\sharp \left\langle \bigsqcup j \ : j \in \mathbb{N} : \ g^\sharp.j \right\rangle \right\rangle$
$\Rightarrow$ { Monotonía de γ }
$\left\langle \forall i \ : i \in \mathbb{N} : \ \gamma.(f^{\sharp(i)}.\bot^\sharp) \sqsubseteq \gamma.\left\langle \bigsqcup j \ : j \in \mathbb{N} : \ g^\sharp.j \right\rangle \right\rangle$
$\Rightarrow$ { Supremo en cpo Γ }
$\left\langle \bigsqcup i \ : i \in \mathbb{N} : \ \gamma.(f^{\sharp(i)}.\bot^\sharp) \right\rangle \sqsubseteq \gamma.\left\langle \bigsqcup j \ : j \in \mathbb{N} : \ g^\sharp.j \right\rangle$
$\Rightarrow$ { Teorema 3.5 y transitividad de $\sqsubseteq$ }
$\mu.f \sqsubseteq \gamma.\left\langle \bigsqcup j \ : j \in \mathbb{N} : \ g^\sharp.j \right\rangle$ □

A partir del resultado de este teorema se puede modificar el método esquematizado en el programa 3.1 para aproximar $\mu.f$. Este nuevo método se presenta en el programa 3.2.

Programa 3.2 Aproximación de $\mu.f$

aproximar_$\mu.f$ ($f^\sharp : \mathcal{P}^\sharp \to \mathcal{P}^\sharp$, $\nabla : \mathcal{P}^\sharp \to \mathcal{P}^\sharp \to \mathcal{P}^\sharp$) : $\mathcal{P}^\sharp$
  $G^\sharp_k := \bot^\sharp$;
  do $\neg(G^\sharp_k \ \nabla \ f^\sharp.G^\sharp_k \ \sqsubseteq^\sharp \ G^\sharp_k)$
    $G^\sharp_k := G^\sharp_k \ \nabla \ f^\sharp.G^\sharp_k$
  od;
  return $G^\sharp_k$

El método calcula la cadena ascendente $g^\sharp.0 \sqsubseteq^\sharp g^\sharp.1 \sqsubseteq^\sharp g^\sharp.2 \sqsubseteq^\sharp \cdots$ del teorema anterior utilizando como función semántica abstracta a $f^\sharp$. Según el teorema la cadena es finita por lo tanto el programa termina.

Cabe agregar que en la práctica, con el fin de mejorar la aproximación de $\mu.f$, se suele comenzar el programa sin aplicar el operador de widening durante un número fijo de iteraciones. De allí en más se aplica el operador hasta que converja en la aproximación. Este método es esquematizado en el programa 3.3.

Programa 3.3 Aproximación de $\mu.f$

```
aproximar_μ.f (f♯ : P♯ → P♯, ∇: P♯ → P♯ → P♯, M : ℕ) : P♯
  k := 0;
  G♯_k := ⊥♯;
  do k < M ∧ ¬(f♯.G♯_k ⊑♯ G♯_k)
    G♯_k := f♯.G♯_k;
    k := k + 1
  od;
  do ¬(G♯_k ∇ f♯.G♯_k ⊑♯ G♯_k)
    G♯_k := G♯_k ∇ f♯.G♯_k
  od;
  return G♯_k
```

Mediante este programa se puede calcular una aproximación del mínimo punto fijo de la función semántica concreta f al cual denotaremos con $(\mu.f)^\sharp$. A partir del teorema 3.7 se puede demostrar $\mu.f \sqsubseteq \gamma.(\mu.f)^\sharp$. Teniendo en mente la función semántica concreta $\mathcal{F}_{\mathcal{T},\Theta}$ como mencionamos en el ejemplo 3.2, podemos enunciar el siguiente teorema el cual nos permite encontrar invariantes de forma automática:

Teorema 3.8 (Generación de invariantes)
Dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$, una abstracción $(\Gamma, \Gamma^\sharp, \gamma)$ con $\Gamma \doteq (\mathsf{Pred}_\mathcal{L}, \subseteq)$ el dominio semántico concreto y $\Gamma^\sharp \doteq (\mathcal{P}^\sharp, \sqsubseteq^\sharp)$ un dominio semántico abstracto. Sea además una función semántica abstracta $f^\sharp : \mathcal{P}^\sharp \to \mathcal{P}^\sharp$ de la función semántica concreta $\mathcal{F}_{\mathcal{T},\Theta} : \mathsf{Pred}_\mathcal{L} \to \mathsf{Pred}_\mathcal{L}$, junto con un operador de widening $\nabla\colon \mathcal{P}^\sharp \to \mathcal{P}^\sharp \to \mathcal{P}^\sharp$. Entonces, la concretización de la aproximación de $(\mu.\mathcal{F}_{\mathcal{T},\Theta})^\sharp$ obtenida por medio del algoritmo 3.3 cumple

$$\mu.\mathcal{F}_{\mathcal{T},\Theta} \subseteq \gamma.(\mu.\mathcal{F}_{\mathcal{T},\Theta})^\sharp$$

y por lo tanto $\gamma.(\mu.\mathcal{F}_{\mathcal{T},\Theta})^\sharp$ es un invariante de TS. □

La demostración de este teorema se deduce, como ya dijimos, del teorema 3.7 y el teorema 2.19 (pág. 49).

3.2 Poliedros convexos cerrados

En la sección anterior vimos uno de los marcos de la teoría de interpretación abstracta, mediante el cual, aplicándolo a un dominio semántico abstracto en particular, nos permite encontrar una aproximación del mínimo punto fijo $\mu.\mathcal{F}_{\mathcal{T},\Theta}$ en un sistema de transiciones. En esta sección desarrollaremos el dominio abstracto utilizado en este trabajo el cual estará basado en la teoría de poliedros convexos cerrados. Los poliedros cerrados son objetos matemáticos presentados generalmente como cuerpos geométricos cuya superficie se compone por una cantidad finita de polígonos. A su vez son convexos cuando para dos puntos pertenecientes a él, existe un segmento de recta interno al poliedro que los une. Esto es más

fácil de entender pensando en “visibilidad entre puntos” [Chv83]: en un poliedro convexo todo punto tiene en su campo de visión a los otros puntos.

Estas entidades matemáticas nos servirán para definir cierta clase de predicados sobre el espacio de estados: se puede pensar a cada variable del programa como una dimensión de $\mathbb{R}^n$ con lo cual n será la cantidad total de variables que aparecen en nuestro programa y un punto en este espacio se identificará con un estado. De esta forma un poliedro cerrado convexo representará el conjunto de estados que encierra, o sea un predicado particular en Pred.

En nuestro caso un programa se representa con un sistemas de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ y sus propiedades se expresan como elementos en $\mathsf{Pred}_{\mathcal{L}}$. Extendiendo la idea del párrafo anterior asociaremos un poliedro a cada locación en $\mathcal{L}$ con lo cual podremos representar de forma abstracta elementos de $\mathsf{Pred}_{\mathcal{L}}$.

A continuación desarrollaremos la teoría básica de poliedros convexos introducida a partir de dos de sus representaciones. Como veremos luego, el manejo de ambas representaciones en paralelo, permite mecanizar de manera mas simple las operaciones necesarias al momento de buscar aproximaciones de $\mu.\mathcal{F}_{\mathcal{T},\Theta}$ sobre este dominio semántico abstracto.

3.2.1 Representación algebraica

De aquí en adelante, supondremos un espacio de estados $\Sigma \doteq \mathsf{Var} \to \mathbb{R}$ donde $\mathsf{Var} \doteq \{x^1, \cdots, x^n\}$ es un conjunto finito de n variables. A continuación veremos una formalización algebraica de la noción intuitiva de poliedro vista al principio de esta sección, la cual nos permitirá denotarlos a través de expresiones.

Definición 3.9 (Desigualdad lineal)
Dada una matriz fila $A \doteq [\, a_i \mid 1 \leqslant i \leqslant n \,] \in \mathbb{R}^{1\times n}$ y un elemento $b \in \mathbb{R}$, llamaremos a la expresión $\left\langle \sum i \, : 1 \leqslant i \leqslant n : \, a_i x^i \right\rangle \leqslant b$, o en forma matricial[2] $AX \leqslant b$ con $X \doteq [\, x^i \mid 1 \leqslant i \leqslant n \,]$, *desigualdad lineal* o simplemente desigualdad. □

Definición 3.10 (Semiespacio cerrado)
El subconjunto de $\mathbb{R}^n$ formado por las soluciones de una desigualdad lineal será llamado *semiespacio cerrado* (o simplemente semiespacio) de $\mathbb{R}^n$. Esto es, una desigualdad lineal $AX \leqslant b$ denotará un semiespacio cerrado $\mathcal{H}$ definido por

$$\mathcal{H} \doteq \{V \in \mathbb{R}^n \mid AV \leqslant b\} \quad .$$ □

Cuando la matriz fila A es distinta de cero el semiespacio cerrado puede pensarse de manera intuitiva como el conjunto de puntos a un lado de un hiperplano perpendicular al vector definido por A.

Definición 3.11 (Convexo)
Un conjunto $\mathcal{C} \subseteq \mathbb{R}^n$ es *convexo* si y solo si para cualesquier par de elementos del conjunto, el segmento que los une está incluido en el mismo:

$$\left\langle \forall\, X_1, X_2 \, : X_1 \in \mathcal{C} \wedge X_2 \in \mathcal{C} : \left\langle \forall\, \lambda \, : \lambda \in [0..1] : \, \lambda X_1 + (1-\lambda) X_2 \in \mathcal{C} \right\rangle \right\rangle \quad .$$

□

[2] Si pensamos a A como un vector en $\mathbb{R}^n$ la desigualdad puede también escribirse con un producto escalar $A \cdot X \leqslant b$.

Los semiespacios cerrados son ejemplos de conjuntos convexos. Se puede demostrar que la propiedad de convexidad es preservada por intersecciones finitas (la intersección de dos conjuntos convexos es convexo) pero no por uniones (la unión de dos conjuntos convexos no es necesariamente convexa).

Con estas definiciones podemos formalizar el concepto de poliedro convexo cerrado.

Definición 3.12 (Poliedro convexo cerrado)
Un *poliedro convexo cerrado* (o poliedro convexo o simplemente poliedro) será una intersección finita de semiespacios cerrados. □

Al conjunto de todos los poliedros convexos cerrados en $\mathbb{R}^n$ se lo denotará como Poly, donde la cantidad de variables n se sobreentenderá del contexto.

A partir de estas definiciones denotaremos un poliedro como una conjunción de desigualdades lineales $A^1X \leqslant b^1 \wedge \cdots \wedge A^mX \leqslant b^m$. La representación de un poliedro $\mathcal{P}$: Poly por un conjunto de desigualdades se escribirá como

$$\mathcal{P} \doteq [\![A^1X \leqslant b^1 \wedge \cdots \wedge A^mX \leqslant b^m]\!]_{\mathrm{r}} \ .$$

Al conjunto de desigualdades lineales utilizado para esta representación lo llamaremos *sistema de restricciones lineales*. Cabe agregar que para simplificar esta representación se pueden incluir igualdades lineales de la forma $AX = b$. De todas maneras, no es necesario incluirlas para definir un poliedro ya que se pueden reemplazar por la conjunción de dos desigualdades.

En este trabajo utilizaremos de forma frecuente la notación matricial para los sistemas de restricciones lineales: si pensamos a $A^1, \cdots, A^m$ como las filas de una matriz $A : \mathbb{R}^{m\times n}$ y a $b^1, \cdots, b^m$ las componentes de un vector columna $B : \mathbb{R}^m$, podemos denotar al poliedro como

$$\mathcal{P} \doteq [\![AX \leqslant B]\!]_{\mathrm{r}} \ ,$$

tal que, si $A^j \doteq [\, a_i^j \mid 1 \leqslant i \leqslant n \,]$ con $j = 1, \cdots, m$, definiremos

$$\begin{aligned} A &\doteq [\, a_i^j \mid 1 \leqslant i \leqslant n \wedge 1 \leqslant j \leqslant m \,] \ , \\ X &\doteq [\, x^i \mid 1 \leqslant i \leqslant n \,] \quad \text{y} \\ B &\doteq [\, b^j \mid 1 \leqslant j \leqslant m \,] \ . \end{aligned}$$

Cuando el vector de variables se deduzca del contexto denotaremos al poliedro también con el par (A, B), esto es

$$\mathcal{P} \doteq [\![(A, B)]\!]_{\mathrm{r}} \ .$$

En particular el poliedro que representa todo $\mathbb{R}^n$ será denotado por un conjunto vacío de restricciones ($m = 0$ en la forma matricial anterior).

3.2.2 Representación geométrica

Como vimos en la sección anterior una forma de representar poliedros convexos es mediante un conjunto de desigualdades lineales. En esta sección veremos otra forma de representación de tipo geométrica. Inicialmente presentaremos la terminología utilizada para denotar elementos de carácter geométrico en $\mathbb{R}^n$.

Definición 3.13 (Combinación lineal)
Sea $\mathcal{V} \doteq \{V^1, \cdots, V^p\}$ un conjunto de vectores en $\mathbb{R}^n$ y $a_1, \cdots, a_p$ escalares en $\mathbb{R}$, un vector de la forma de la forma $\left\langle \sum i \ : 1 \leqslant i \leqslant p : \ a_i V^i \right\rangle$ será llamado una *combinación lineal* de los vectores $\mathcal{V}$. El conjunto vectores formado por todas las combinaciones lineales de $\mathcal{V}$ será llamado *subespacio lineal generado* por el conjunto de vectores. □

Definición 3.14 (Envoltura cónica)
Dado un conjunto finito de vectores $\mathcal{V} \doteq \{V^1, \cdots, V^p\}$ y $\mu_1, \cdots, \mu_p$ escalares en $\mathbb{R}^+$ (números reales positivos) un vector de la forma $\left\langle \sum i \ : 1 \leqslant i \leqslant p : \ \mu_i V^i \right\rangle$ será llamado una *combinación lineal positiva* de los vectores $\mathcal{V}$. El conjunto de todas las combinaciones positivas de un conjunto $\mathcal{V}$ será llamado *envoltura cónica*[3] de $\mathcal{V}$. □

Una intuición geométrica de este conjunto es pensarlo como el mínimo cono proyectado al infinito, con vértice en el origen de coordenadas cartesianas y que contiene todos los puntos de $\mathcal{V}$.

Definición 3.15 (Envoltura convexa)
Dado un conjunto finito de vectores $\mathcal{V} \doteq \{V^1, \cdots, V^p\}$ y escalares $\lambda_1, \cdots, \lambda_p$ pertenecientes a $\mathbb{R}^+$ tal que $\langle \sum i \ : 1 \leqslant i \leqslant p : \ \lambda_i \rangle = 1$, un vector de la forma $\left\langle \sum i \ : 1 \leqslant i \leqslant p : \ \lambda_i V^i \right\rangle$ será llamado *combinación lineal convexa* de los vectores $\mathcal{V}$. El conjunto de todas las combinaciones convexas de $\mathcal{V}$ será llamado *envoltura convexa*[4] de $\mathcal{V}$. □

Una intuición geométrica de este conjunto es pensarlo como el mínimo poliedro que contiene los puntos de $\mathcal{V}$.

Definición 3.16 (Vértice)
Un *vértice* de un poliedro convexo $\mathcal{P} \in \mathsf{Poly}$ será un vector $S \in \mathcal{P}$ que no es combinación lineal convexa de otros vectores en $\mathcal{P}$. Esto es, para todo conjunto finito de vectores $\mathcal{V} \doteq \{V^i, \cdots, V^p\}$ con $\mathcal{V} \subseteq \mathcal{P}$ y para todo conjunto de escalares positivos $\lambda_1, \cdots, \lambda_p \in \mathbb{R}^+$ tal que $\langle \sum i \ : 1 \leqslant i \leqslant p : \ \lambda_i \rangle = 1$ entonces

$$\left\langle \forall i \ : 1 \leqslant i \leqslant p : \ \lambda_i = 0 \ \vee \ V^i = S \right\rangle \ . \qquad \square$$

Puede demostrarse que un poliedro contiene solo una cantidad finita de vértices.

Definición 3.17 (Rayo)
Dado un poliedro convexo no vacío $\mathcal{P} \in \mathsf{Poly}$, un *rayo* del mismo será un vector no nulo R tal que existe una semirrecta paralela al vector incluida en $\mathcal{P}$, esto es

$$\langle \forall \mu, V \ : \mu \in \mathbb{R}^+ \wedge V \in \mathcal{P} : \ V + \mu R \in \mathcal{P} \rangle \ .$$

Se dirá que dos rayos R y R' son iguales si ambos vectores son dependientes y tienen la misma dirección, esto es si $\langle \exists \mu \ : \mu \in \mathbb{R}^+ : \ R = \mu R' \rangle$.

Un *rayo extremo* R de un poliedro $\mathcal{P}$ será un rayo que no es combinación lineal positiva de otros rayos del poliedro. Esto es, para todo conjunto finito de rayos $\mathcal{R} \doteq \{R^i, \cdots, R^p\}$ en el poliedro y para todo conjunto de escalares positivos $\mu_1, \cdots, \mu_p \in \mathbb{R}^+$ tales que $R = \left\langle \sum i \ : 1 \leqslant i \leqslant p : \ \mu_i R^i \right\rangle$ entonces

$$\left\langle \forall i \ : 1 \leqslant i \leqslant p : \ \mu_i = 0 \ \vee \ R^i = R \right\rangle \ . \qquad \square$$

[3] *Conical hull* en inglés.
[4] *Convex hull* en inglés

De manera intuitiva si un rayo extremo es trasladado sobre un vértice, la semirrecta desde ese vértice hacia la dirección del rayo coincide con un borde de una cara del poliedro. Con esta definición se pueden definir unívocamente los rayos de un poliedro módulo la igualdad entre ellos definida anteriormente. Puede demostrarse que un poliedro contiene solo una cantidad finita de rayos extremos.

Definición 3.18 (Línea)
Dado un poliedro no vacío $\mathcal{P} \in \mathsf{Poly}$, una *línea* será un vector no nulo D tal que $-D$ y el mismo son rayos del poliedro, esto es

$$\left\langle \forall u, V \ : u \in \mathbb{R} \wedge V \in \mathcal{P} : \ V + uD \in \mathcal{P} \right\rangle \ . \qquad \square$$

Un poliedro que tenga al menos una línea será llamado *cilindro*. Diremos que dos líneas son iguales si son linealmente dependientes.

Para poder representar un poliedro con vértices rayos y líneas necesitaremos también de los siguientes conceptos.

Definición 3.19 (Variedad lineal)
Un subconjunto de un espacio vectorial resultante del traslado de un subespacio vectorial por un vector será llamado *variedad lineal*[5]. Esto es, dado $\mathcal{E}$ un espacio vectorial, $\mathcal{M} \subseteq \mathcal{E}$ es una variedad lineal si existe un subespacio $\mathcal{L} \subseteq \mathcal{E}$ y un elemento $V \in \mathcal{E}$ tal que $\mathcal{M} = \{V + L \mid L \in \mathcal{L}\}$. $\square$

Notar que bajo esta definición el subespacio $\mathcal{L}$ está definido unívocamente mientras V lo está solamente módulo $\mathcal{L}$, esto es si $\mathcal{M} = \mathcal{M}'$ con $\mathcal{M} = \{V + L \mid L \in \mathcal{L}\}$ y $\mathcal{M}' = \{V' + L \mid L \in \mathcal{L}'\}$ entonces $\mathcal{L} = \mathcal{L}'$ y $V - V' \in \mathcal{L}$. A partir de esta observación, la *dimensión de la variedad lineal* $\mathcal{M}$ puede definirse como la dimensión de $\mathcal{L}$. Si la dimensión del espacio vectorial $\mathcal{E}$ es n y la de una variedad lineal $\mathcal{M} \subseteq \mathcal{E}$ es $n-1$ diremos que $\mathcal{M}$ es un *hiperplano* y si la dimensión es 1 diremos que es una *recta*. De esta forma también podemos definir línea como una variedad de dimensión 1. La *dimensión de un poliedro convexo* $\mathcal{P}$ será definida como la dimensión de la menor variedad lineal que contiene a $\mathcal{P}$.

Dada una variedad lineal $\mathcal{M} \doteq \{V + L \mid L \in \mathcal{L}\}$ con el subespacio $\mathcal{L}$ generado por una base $\mathcal{B} \doteq \left\{B^i \in \mathcal{L} \mid 1 \leqslant i \leqslant m \leqslant n\right\}$ diremos que la variedad $\mathcal{M}$ está generada por $(V, \mathcal{B})$. Además, una variedad lineal definida como $\mathcal{M}^\perp \doteq \left\{V + L \mid L \in \mathcal{L}^\perp\right\}$ con $\mathcal{L}^\perp$ el subespacio ortogonal a $\mathcal{L}$, será llamada *variedad ortogonal* a $\mathcal{M}$.

Se puede demostrar que cualquier variedad lineal contenida en un poliedro, que es un cilindro, está contenida en una variedad $\mathcal{M} \doteq \{V + L \mid L \in \mathcal{L}\}$ con $\mathcal{L}$ el subespacio vectorial generado por las líneas del cilindro y $V \in \mathcal{P}$ un punto del mismo. En este sentido se dice que las variedades lineales generadas por las líneas de un poliedro son *maximales* con respecto a las contenidas en el poliedro.

A partir de la definición de vértices, rayos y líneas vamos a construir una representación de poliedros convexos, comenzando con casos particulares hasta llegar a poliedros convexos generales.

Un *poliedro acotado* será aquel que no tiene líneas ni rayos. Puede demostrarse que cada punto del mismo es una combinación lineal convexa de sus vértices,

[5] También llamado *espacio afín*.

por lo cual el poliedro es la envoltura convexa de los mismos. De esta forma un poliedro acotado puede ser denotado por su conjunto de vértices únicamente.

Los puntos de un poliedro que no posee líneas pueden expresarse como una combinación lineal de sus vértices más una combinación lineal positiva de sus rayos extremos. Esto es, dado $\mathcal{P}$ un poliedro de esta clase, con $\mathcal{S} \doteq \{S^1, \cdots, S^p\}$ sus vértices y $\mathcal{R} \doteq \{R^1, \cdots, R^q\}$ sus rayos extremos, si $X \in \mathcal{P}$ entonces existen escalares positivos $\lambda_1, \cdots, \lambda_p, \mu_1, \cdots, \mu_q \in \mathbb{R}^+$ con $\langle \sum i \,:\, 1 \leqslant i \leqslant p : \lambda_i \rangle = 1$ tal que

$$X = \left\langle \sum i \,:\, 1 \leqslant i \leqslant p : \lambda_i S^i \right\rangle + \left\langle \sum i \,:\, 1 \leqslant i \leqslant q : \mu_i R^i \right\rangle$$

o de igual forma

$$\begin{aligned}\mathcal{P} = \{\left\langle \sum i \,:\, 1 \leqslant i \leqslant p : \lambda_i S^i \right\rangle + \left\langle \sum i \,:\, 1 \leqslant i \leqslant q : \mu_i R^i \right\rangle \mid \\ \lambda_1, \cdots, \lambda_p, \mu_1, \cdots, \mu_q \in \mathbb{R}^+ \wedge \left\langle \sum i \,:\, 1 \leqslant i \leqslant p : \lambda_i \right\rangle = 1\}\end{aligned}$$

Con este resultado, un poliedro sin líneas puede denotarse con su conjunto de vértices y rayos extremos asociados.

Para el caso general en que el poliedro posea líneas (sea un cilindro) procederemos de la siguiente forma: dado un poliedro convexo $\mathcal{P}$ tomaremos una variedad lineal $\mathcal{M}^\perp$ ortogonal a la definida por sus líneas. La intersección de esta variedad con el poliedro será llamada una *sección* del poliedro. Puede demostrarse que una sección no posee líneas y lo que es más importante, cualquier punto del poliedro es igual a la suma de una una combinación convexa de los vértices de su sección, una combinación lineal positiva de sus rayos y una combinación lineal de las líneas del poliedro. Esto es, tomamos una base $\mathcal{D} \doteq \{D^1, \cdots, D^t\}$ del subespacio generado por las lineas del poliedro, al cual llamaremos $\mathcal{L}_\mathcal{D}$. A partir del mismo se obtiene su subespacio ortogonal $\mathcal{L}_\mathcal{D}^\perp$ y una variedad de la forma $\mathcal{M}^\perp \doteq \left\{V + L \mid L \in \mathcal{L}_\mathcal{D}^\perp\right\}$. El poliedro convexo definido como $\mathcal{P}' \doteq \mathcal{P} \cap \mathcal{M}^\perp$ será una sección del mismo. Sean $\mathcal{S} \doteq \{S^1, \cdots, S^p\}$ los vértices y $\mathcal{R} \doteq \{R^1, \cdots, R^q\}$ los rayos extremos de $\mathcal{P}'$, entonces para todo elemento $X \in \mathcal{P}$ del poliedro existen escalares positivos $\lambda_1, \cdots, \lambda_p, \mu_1, \cdots, \mu_q \in \mathbb{R}^+$ con $\langle \sum i \,:\, 1 \leqslant i \leqslant p : \lambda_i \rangle = 1$ y escalares arbitrarios $\nu_1, \cdots, \nu_t \in \mathbb{R}$ tal que

$$\begin{aligned}X = &\left\langle \sum i \,:\, 1 \leqslant i \leqslant p : \lambda_i S^i \right\rangle + \left\langle \sum i \,:\, 1 \leqslant i \leqslant q : \mu_i R^i \right\rangle \\ &+ \left\langle \sum i \,:\, 1 \leqslant i \leqslant t : \nu_i D^i \right\rangle\end{aligned}$$

con lo cual el poliedro $\mathcal{P}$ puede expresarse como el siguiente conjunto

$$\begin{aligned}\mathcal{P} = \{ &\left\langle \sum i \,:\, 1 \leqslant i \leqslant p : \lambda_i S^i \right\rangle + \left\langle \sum i \,:\, 1 \leqslant i \leqslant q : \mu_i R^i \right\rangle \\ &+ \left\langle \sum i \,:\, 1 \leqslant i \leqslant t : \nu_i D^i \right\rangle \\ &| \lambda_1, \cdots, \lambda_p, \mu_1, \cdots, \mu_q \in \mathbb{R}^+ \\ &\wedge \nu_1, \cdots, \nu_q \in \mathbb{R} \wedge \left\langle \sum i \,:\, 1 \leqslant i \leqslant p : \lambda_i \right\rangle = 1 \} \ .\end{aligned}$$

La 3-upla $(\mathcal{S}, \mathcal{R}, \mathcal{D})$ será llamada *marco*[6] del poliedro. La parte derecha de la última ecuación será llamada *envoltura convexa de un marco*. La misma muestra que el poliedro se genera obteniendo primero su sección (a partir de sus vértices y rayos extremos) y trasladando la misma sobre sus líneas.

[6] Frame en inglés.

A partir de este resultado cualquier poliedro convexo puede denotarse con su conjunto de vértices, rayos extremos y líneas asociadas. Si $(\mathcal{S}, \mathcal{R}, \mathcal{D})$ representa el poliedro $\mathcal{P}$ escribiremos:

$$\mathcal{P} \doteq [\![(\mathcal{S}, \mathcal{R}, \mathcal{D})]\!]_{\mathrm{m}} \ .$$

El caso particular del poliedro vacío será representado por la tupla con tres conjuntos vacíos.

En función de lo expuesto en esta sección y en la anterior, un poliedro convexo cerrado puede ser representado de dos formas:

- como el par (A, B) con lo cual el poliedro será el conjunto de soluciones del sistema de restricciones lineales $AX \leqslant B$ y
- como un marco $(\mathcal{S}, \mathcal{R}, \mathcal{D})$ con lo cual el poliedro será su envoltura convexa.

En este trabajo se utilizarán simultáneamente las dos representaciones de un poliedro. La necesidad de esta doble representación se debe a que algunas operaciones sobre poliedros son fácilmente implementables solo en una de ellas. A continuación se presentarán algoritmos para hacer la conversión entre ambas representaciones.

3.2.3 Conversión de marco a sistema de restricciones

Dado un poliedro $\mathcal{P}$ junto con su marco $(\mathcal{S}, \mathcal{R}, \mathcal{D})$, esta conversión consiste en encontrar el sistema que lo represente. A continuación se dará una idea general del método para obtenerlas. Más adelante profundizaremos en su implementación.

Sea $(\mathcal{S}, \mathcal{R}, \mathcal{D})$ el marco de un poliedro $\mathcal{P}$ no vacío ($\mathcal{P} \neq \varnothing$), donde

$$\mathcal{S} \doteq \{S^1, \cdots, S^p\}, \ \ \mathcal{R} \doteq \{R^1, \cdots, R^q\}, \ \ \mathcal{D} \doteq \{D^1, \cdots, D^t\} \ .$$

Por lo expuesto en la subsección anterior, cada punto $X \doteq [x^i | 1 \leqslant i \leqslant n]$ perteneciente a un poliedro $\mathcal{P}$ es caracterizado por la existencia de coeficientes λ_i, μ_j, ν_k en $\mathbb{R}$ tal que:

$$\begin{aligned} X = & \left\langle \textstyle\sum i \ : 1 \leqslant i \leqslant p : \ \lambda_i S^i \right\rangle + \left\langle \textstyle\sum i \ : 1 \leqslant i \leqslant q : \ \mu_i R^i \right\rangle \\ & + \left\langle \textstyle\sum i \ : 1 \leqslant i \leqslant t : \ \nu_i D^i \right\rangle \end{aligned}$$

con

$$\begin{aligned} & \lambda_i \geqslant 0, \qquad 1 \leqslant i \leqslant p \\ & \mu_i \geqslant 0, \qquad 1 \leqslant i \leqslant q \\ & \left\langle \textstyle\sum i \ : 1 \leqslant i \leqslant p : \ \lambda_i \right\rangle = 1 \ . \end{aligned}$$

Por lo tanto, con la envoltura convexa del marco se puede caracterizar al poliedro $\mathcal{P}$ con una expresión existencial sobre un sistema de restricciones en $\mathbb{R}^{n+p+q+t}$ con variables ligadas $\lambda_1, \cdots, \lambda_p$, $\mu_1, \cdots, \mu_q$ y $\nu_1, \cdots, \nu_t$. Puede demostrarse que una expresión cuantificada de este tipo es equivalente (representa el mismo poliedro) a un sistema de restricciones lineales con el cuantificador removido. Además, este sistema equivalente puede ser obtenido proyectando el sistema de restricciones dentro de la cuantificación sobre las dimensiones no cuantificadas, es decir, el poliedro cuantificado en $\mathbb{R}^{n+p+q+t}$ se proyecta en $\mathbb{R}^n$

eliminando de esta forma la cuantificación. Este método de eliminación fue introducido por Fourier en 1827 [Fou41] y descripto por T. S. Motzkin [Mot36]. Una versión del mismo es descripta en [Kuh56] en la forma de una aplicación sucesiva de la operación de *proyección* (que se explicará en los próximos párrafos), sobre las variables λ_i, μ_j, ν_k obteniéndose así un sistema de restricciones lineales en $\mathbb{R}^n$ que representa al poliedro. Este método será presentado a continuación.

Proyección

Sea $AX \leqslant B$ un sistema de m desigualdades, donde A es una matriz en $\mathbb{R}^{m\times n}$, B es un vector columna en $\mathbb{R}^m$, y X es el vector de variables, tal como lo muestra el siguiente gráfico:

$$\underbrace{\begin{pmatrix} a_1^1 & a_2^1 & \cdots & a_n^1 \\ a_1^2 & a_2^2 & \cdots & a_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ a_1^m & a_2^m & \cdots & a_n^m \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} x^1 \\ x^2 \\ \vdots \\ x^n \end{pmatrix}}_{X} \leqslant \underbrace{\begin{pmatrix} b^1 \\ b^2 \\ \vdots \\ b^m \end{pmatrix}}_{B}$$

Para eliminar la variable x_l, se construye un nuevo sistema (una “proyección” del anterior sobre la l-ésima columna) de la siguiente forma:

- Cada fila A^h de A tal que $a_l^h = 0$, la desigualdad $A^hX \leqslant b^h$ es parte del nuevo sistema.
- Para dos filas A^{h_1}, A^{h_2} de A tal que $a_l^{h_1}.a_l^{h_2} < 0$ (tienen distinto signo), la restricción construida por $(|a_l^{h_1}|.A^{h_2} + |a_l^{h_2}|.A^{h_1}).X \leqslant |a_l^{h_1}|.b^{h_2} + |a_l^{h_2}|.b^{h_1}$ es parte del nuevo sistema.

El sistema resultante contiene sólo 0’s en la l-ésima columna, con lo cual esa columna es independiente de x_l y, por ende, puede ser eliminada. De manera intuitiva, esta operación se puede entender como una proyección en la dirección del eje l al hiperplano definido sobre las demás dimensiones.

Utilizando esta operación, en el próximo apartado se detalla el procedimiento a seguir para lograr obtener la envoltura convexa de un marco.

Envoltura convexa de un marco

Para calcular la envoltura convexa de un marco $(\mathcal{S}, \mathcal{R}, \mathcal{D})$ correspondiente a un poliedro $\mathcal{P}$, se obtendrán sucesivos sistemas de restricciones lineales denotados por los pares $(A_1, B_1), \cdots, (A_{p+q+t}, B_{p+q+t})$ que representarán poliedros $\mathcal{P}_1, \cdots, \mathcal{P}_{p+q+d}$ respectivamente. Inicialmente, el poliedro $\mathcal{P}_1$ tendrá como marco solo el primer vértice S_1 a partir del cual se calcula su sistema de restricciones. Desde aquí, para calcular el sistema de restricciones correspondiente al poliedro $\mathcal{P}_{i+1}$, el algoritmo incorporará sucesivamente un vértice, rayo o línea del marco $(\mathcal{S}, \mathcal{R}, \mathcal{D})$ al marco del poliedro $\mathcal{P}_i$. De esta forma, un poliedro $\mathcal{P}_i$ de la secuencia será la envoltura convexa de los primero i elementos del marco $(\mathcal{S}, \mathcal{R}, \mathcal{D})$. Finalmente el poliedro $\mathcal{P}_{p+q+t}$ será igual a $\mathcal{P}$ y el par (A_{p+q+t}, B_{p+q+t}) denotará al sistema de restricciones $A_{p+q+t}X \leqslant B_{p+q+t}$ cuyo conjunto de soluciones será el poliedro.

Como ya dijimos, el cálculo de las restricciones del primer poliedro será el sistema de restricciones denotado por $(X = S_1)$. Al escribir este sistema en forma matricial tendremos A_1 como una matriz $2n \times n$ y B_1 un vector columna con $2n$ filas. A continuación se detalla este resultado:

$$(X = S_1) \equiv (X \leqslant S_1 \wedge -X \leqslant -S_1) \ .$$

Expandiendo X y S_1, y separando A_1, se obtiene:

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ -x_1 \\ -x_2 \\ \vdots \\ -x_n \end{pmatrix} \leqslant \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \\ -s_1 \\ -s_2 \\ \vdots \\ -s_n \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ -1 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & -1 \end{pmatrix}}_{A_1} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \leqslant \underbrace{\begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \\ -s_1 \\ -s_2 \\ \vdots \\ -s_n \end{pmatrix}}_{B_1}$$

Con el sistema de restricciones $A_1X \leqslant B_1$ de $\mathcal{P}_1$ ya obtenido, para calcular cada sistema correspondiente a $\mathcal{P}_i$ $(1 < i \leqslant p + q + t)$ se procede a obtener los resultados de las siguientes funciones:

$$(A_i, B_i) = \begin{cases} \textbf{addVertex}(A_{i-1}X \leqslant B_{i-1}, S_i) & 1 < i \leqslant p & (1) \\ \textbf{addRay}(A_{i-1}X \leqslant B_{i-1}, R_{i-p}) & p < i \leqslant p + q & (2) \\ \textbf{addLine}(A_{i-1}X \leqslant B_{i-1}, D_{i-p-q)}) & p + q < i \leqslant p + q + t & (3) \end{cases}$$

El primer paso consiste en obtener (1). Se puede demostrar que cualquier punto en el nuevo poliedro es una combinación convexa de otros dos puntos del poliedro anterior:

$$X \in \mathcal{P}_i \equiv \left\langle \exists\, X', \lambda \ : X \in \mathcal{P}_{i-1} \wedge 0 \leqslant \lambda \leqslant 1 : \ X = \lambda X' + (1 - \lambda) S_i \right\rangle$$

y como cualquier punto del poliedro anterior es solución de su sistema de restricciones se puede eliminar la variable cuantificada:

$$\left\langle \exists\, \lambda \ : 0 \leqslant \lambda \leqslant 1 : \ A_{i-1}X + \lambda(A_{i-1}S_i - B_{i-1}) \leqslant A_{i-1}S_i \right\rangle \ .$$

Esta ecuación puede escribirse como un sistema de desigualdades con la variable nueva λ. Con ello, si A_{i-1} es de dimensión $m \times n$ y A_i es de dimensión $(m + 2) \times (n + 1)$. Realizando una proyección sobre λ, se obtiene un nuevo sistema de m' restricciones y n columnas (una columna por variable).

En (2) el procedimiento es análogo: se van agregando los rayos creando combinaciones positivas con las restricciones ya obtenidas en los pasos anteriores. Es decir que el sistema de restricciones del poliedro $\mathcal{P}_i$, $i \in \{\sigma + 1 \ldots \sigma + \rho\}$ es:

$$\left\langle \exists\, \mu \ : \mu \in \mathbb{R}^+ : \ A_{i-1}X - \mu A_{i-1}R_{i-p} \leqslant B_{i-1} \right\rangle \ .$$

Aplicando proyección sobre μ en cada iteración se mantiene el sistema de restricciones del paso i-ésimo con n variables.

Finalmente, en (3) se agregan todas las líneas de una en una creando combinaciones lineales con las restricciones del paso anterior (en cada paso se proyecta el sistema sobre ν):

$$\left\langle \exists \, \nu \ : \nu \in \mathbb{R} : \ A_{i-1}X - \nu A_{i-1} D_{i-(p+q)} \leqslant B_{i-1} \right\rangle \ .$$

Simplificación

En general, el sistema de restricciones lineales que se obtiene luego de estas operaciones de proyección contiene un gran número de restricciones irrelevantes, las cuales pueden (y por cuestiones de eficiencia deben) ser eliminadas, siendo esta eliminación invariante para el poliedro que ellas representan.

Antes de mostrar el método de simplificación vamos a definir el siguiente concepto:

Definición 3.20 (Saturación)
Dada una desigualdad $\left\langle \sum i \ : 1 \leqslant i \leqslant n : \ a_i x^i \right\rangle \leqslant b$ diremos que un vértice $S \doteq [\, s^i \mid 1 \leqslant i \leqslant n \,]$ *satura* a la misma si $\left\langle \sum i \ : 1 \leqslant i \leqslant n : \ a_i s^i \right\rangle = b$. De manera intuitiva, un vértice satura una desigualdad cuando se encuentra en el borde del semiespacio que ella define.
También diremos que un rayo $R \doteq [\, r^i \mid 1 \leqslant i \leqslant n \,]$ satura la desigualdad si $\left\langle \sum i \ : 1 \leqslant i \leqslant n : \ a_i r^i \right\rangle = 0$. Esto es, un rayo satura una desigualdad cuando es paralelo al borde del semiespacio que ella define.
De igual manera, una línea $D \doteq [\, d^i \mid 1 \leqslant i \leqslant n \,]$ satura la desigualdad si $\left\langle \sum i \ : 1 \leqslant i \leqslant n : \ a_i d^i \right\rangle = 0$. Es decir, una línea satura una desigualdad cuando es paralela al borde del semiespacio que ella define. □

Conociendo el marco del poliedro, el sistema de restricciones lineales correspondiente puede ser simplificado de acuerdo a los siguientes resultados [Lan66]:

- Una desigualdad que nunca es saturada por un vértice del marco es irrelevante.
- Una desigualdad que es saturada por todos los vértices y todos los rayos del marco representa una igualdad, y todas las igualdades se encuentran de esta forma.
- Dadas dos desigualdades C^i, C^j, se define una relación de quasi-orden entre ellas de la siguiente manera:

  $C^i \sqsubseteq C^j$ si y solo si C^j es saturada por cada vértice y rayo que satura a C^i.

  A partir de esta relación se establece:

  - Si $C^i \sqsubseteq C^j$ pero $C^j \not\sqsubseteq C^i$, C^i es irrelevante.
  - Si $C^i \sqsubseteq C^j$ y $C^j \sqsubseteq C^i$, una y sólo una de las desigualdades es irrelevante y puede ser excluida.

Aplicando estos resultados al sistema de restricciones se obtiene un sistema minimal (sin desigualdades irrelevantes), igual al poliedro con el cual se inició la simplificación. Notar además, que para realizar el procedimiento descripto es necesario conocer ambas representaciones.

3.2.4 Conversión de sistema de restricciones a marco

Lo que se busca en esta conversión es obtener todos los vértices, rayos y líneas de un poliedro representado por un sistema de restricciones lineales[7]. En la actualidad existen diversos métodos para resolver esta conversión, y se categorizan en dos grandes clases: los *métodos de pivoting* y los de *no-pivoting*.

Los métodos de pivoting derivan del método simplex de la programación lineal ([Chv83, BJS90]), el cual encuentra nuevos vértices adyacentes a vértices ya conocidos, usando operaciones simples de pivot. En [CH78] se sugieren varios métodos de pivoting para resolver la conversión del sistema de restricciones al marco de un poliedro. En particular, se detalla el método de Lanery [Lan66], brevemente explicado en el próximo apartado.

Conceptos básicos de programación lineal

Sea $AX \leqslant B$ un sistema de m restricciones lineales y n variables (como los usados en las secciones anteriores). Claramente, este sistema puede ser representado agregando m variables $x^{n+1}, \cdots, x^{n+m}$ y escribiéndolo de la forma:

$$\left\langle \sum i \ : 1 \leqslant i \leqslant n : \ a_i^j x^i \right\rangle + x^{n+j} = b^j \wedge x^{n+j} \geqslant 0 \quad \text{donde} \quad 1 \leqslant j \leqslant m$$

De esta manera, las soluciones sobre las variables $x^1, \cdots, x^n$ en este sistema serán las mismas que en el anterior. Escrito en forma matricial el sistema tiene la forma:

$$\underbrace{\begin{pmatrix} a_1^1 & a_2^1 & \cdots & a_n^1 \\ a_1^2 & a_2^2 & \cdots & a_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ a_1^m & a_2^m & \cdots & a_n^m \end{pmatrix}}_{A} \underbrace{\begin{matrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{matrix}}_{I} \begin{pmatrix} x^1 \\ x^2 \\ \vdots \\ x^n \\ x^{n+1} \\ x^{n+2} \\ \vdots \\ x^{n+m} \end{pmatrix} = \underbrace{\begin{pmatrix} b^1 \\ b^2 \\ \vdots \\ b^m \end{pmatrix}}_{B}$$

$$\wedge \begin{pmatrix} x^{n+1} \\ x^{n+2} \\ \vdots \\ x^{n+m} \end{pmatrix} \geqslant \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

Esta forma de escribir un sistema de restricciones será de importancia a la hora de implementar el método de Lanery. De manera general, la misma puede definirse como:

Definición 3.21 (Forma estándar)
Un sistema de restricciones está escrito en *forma estándar* si es de la forma $AX = B \wedge X^E \geqslant 0$, donde $A \in \mathbb{R}^{m\times(n+m)}$, $X \in \mathsf{Var}^{(n+m)}$, $B \in \mathbb{R}^m$, $E \subseteq [1..(n+m)]$ con $\mathsf{Card}.E = m$ y $X^E \in \mathbb{R}^m$ denota la matriz columna

[7] Un problema afín es el llamado *problema del convex hull*, el cual computa las facetas del convex hull que rodea un conjunto de puntos dado.

$[x^i \mid i \in E]$. Las variables $\{x^i \mid i \in E\}$ se denominan *variables aumentadas*, y las variables $\{x^i \mid i \in F\}$ con $F = [1..(n+m)] - E$ se llaman *variables iniciales*. □

En la ecuación matricial anterior las variables X^F con $F = \{1, \ldots, n\}$ serán las variables iniciales, mientras que las variables X^E con $E = \{n+1, \ldots, n+m\}$ serán las aumentadas. Notemos que las últimas m columnas de la primer matriz forman la matriz identidad. Generalizando este hecho podemos expresar las siguientes definiciones:

Definición 3.22 (Base)
Dado un sistema en forma estándar $AX = B \wedge X^E \geqslant 0$ con $A \in \mathbb{R}^{m \times (n+m)}$, una *base* es una sub-matriz invertible (no singular) A_I de A de tamaño $m \times m$, donde I es un conjunto de índices de columnas de A con $\mathsf{Card}.I = m$ e $I \subseteq [1..(n+m)]$. De esta manera A_I esta formado por las columnas I de la matriz A. □

Aprovechando el hecho que la base A_I es invertible, el sistema puede denotarse de forma equivalente como muestra la siguiente definición:

Definición 3.23 (Forma canónica)
Dado un sistema escrito en forma forma estándar $AX = B \wedge X^E \geqslant 0$ con $A \in \mathbb{R}^{m \times (n+m)}$, su *forma canónica* con respecto a una base A_I es:

$$X^I + ({A_I}^{-1} A_J) X^J = {A_I}^{-1} B \;\wedge\; X^E \geqslant 0$$

donde $J = \{1, \ldots, n+m\} - I$. Se dice además que las variables en X^I están en la base. □

Definición 3.24 (Base factible y adyacencia)
La base A_I es *factible* si y sólo si $({A_I}^{-1} B)^E \geqslant 0$. Dos bases factibles A_I y A_J son *adyacentes* si y sólo si $|I \cap J| = m - 1$. □

Si dos bases son adyacentes, la operación clásica de *pivot* [Chv83] transforma un sistema en forma canónica con respecto a A_I en uno equivalente en forma canónica con respecto a A_J.

El *método de base artificial*, el cual es la inicialización del *método simplex*, transforma el sistema de restricciones en forma estándar en otro equivalente en forma canónica con respecto a una base factible, siempre que esa base exista. Después de aplicar este paso de inicialización, el método propuesto por Lanery [Lan66] nos permite obtener los vértices, rayos y líneas del poliedro definido por el sistema de restricciones lineales.

Antes de comenzar a explicar el método de Lanery, daremos algunas definiciones geométricas:

Definición 3.25 (Cara de un poliedro)
Una *cara de un poliedro* $\mathcal{P}$ será un poliedro convexo $\mathcal{C} \subseteq \mathcal{P}$ tal que para todo punto X contenido en $\mathcal{C}$, los segmentos que contienen a X y están incluidos en $\mathcal{P}$ también lo están en $\mathcal{C}$, esto es:

$$\begin{array}{l} \langle \forall\, X, X_1, X_2, \lambda \;: \\ \quad X \in \mathcal{C} \wedge X_1, X_2 \in \mathcal{P} \wedge \lambda \in [0..1] \wedge X = \lambda X_1 + (1-\lambda) X_2 \;:\; X_1, X_2 \in \mathcal{C} \,\rangle \;. \end{array}$$

Una cara de un poliedro de dimensión k será llamada una *k-cara*. Una arista de un poliedro será una 1-cara. Un vértice de un poliedro que no contenga líneas serán una 0-cara. □

Definición 3.26 (Adyacencia)
Diremos que dos *vértices* pertenecientes al marco de un poliedro son *adyacentes* si y sólo si pertenecen a una misma cara.

En el mismo sentido diremos que dos *rayos* del marco de un poliedro son *adyacentes* si y sólo si son paralelos a una misma 2-cara.

Finalmente, un *vértice* y un *rayo* del marco de un poliedro serán *adyacentes* si y sólo si el vértice pertenece a una arista del poliedro paralela al rayo. En otras palabras, el vértice se encuentra en el extremo de una arista paralela al rayo. □

Principios del método de Lanery

Se puede demostrado que el grafo de relación de adyacencia sobre el conjunto $\{A_I\}$ de bases factibles, que contiene a todas las variables iniciales (es decir, tal que $F \subset I$), es conexo. Por lo tanto, dada una base de $\{A_I\}$, se le puede aplicar sucesivamente operaciones de pivot más una técnica de recorrido exhaustiva para encontrar todas las bases factibles de un sistema de restricciones dado. A partir de este hecho, el método de Lanery se basa en las siguientes propiedades:

- Si A_I es una base factible tal que $F \subset I$, entonces el vector $({A_I}^{-1}B)^F$ corresponde a un vértice del poliedro convexo definido por el sistema de restricciones inicial. A cada vértice de un poliedro que no contiene líneas le corresponde al menos una base factible. Si dos vértices son adyacentes, entonces hay dos bases factibles adyacentes correspondientes a cada uno de los vértices.

- Sea $AX = B$, $X^E \geqslant 0$, la forma canónica de un sistema de restricciones lineales, con respecto a la base factible A_I, la cual contiene todas las variables iniciales (mismo caso anterior). Lanery muestra que si una columna $i_0 \in (E - I)$ satisface la condición

$$\left\langle \forall\, j \;:\, j \in [1..m] \wedge a^j_{i_0} > 0 :\, \left\langle \forall\, k \;:\, k \in (E \cap I) :\; a^j_k = 0 \right\rangle \right\rangle \qquad (3.1)$$

  entonces el vector $R \in \mathbb{R}^n$ definido por:

$$R \doteq [\, r_i \mid r_i = -a^j_{i_0} \wedge a^j_i = 1\,]$$

  es un rayo extremo del poliedro. Aplicando este resultado a cada columna i_0 que satisface la ecuación 3.1, en cada base factible correspondiente a cada vértice S, se encuentran todos los rayos extremos adyacentes a S.

- Para el caso de los poliedros que contienen al menos una línea, se sabe que no hay bases factibles que contengan a todas las variables iniciales. Sea $AX = B$, $X^E \geqslant 0$ la forma canónica con respecto a cualquier base factible A_I. Sea $i_0 \in (F - I)$ una columna que satisface

$$\left\langle \forall\, j, k \;:\, j \in [1..m] \wedge k \in I \wedge a^j_{i_0} \neq 0 :\; a^j_k = 0 \;\vee\; k \in F \right\rangle \;. \qquad (3.2)$$

  Sea D_{i_0} el vector en $\mathbb{R}^n$ tal que la i-ésima coordenada está definida por:

$$(D_{i_0})^i = \begin{cases} 1 & \text{si } i = i_0 \\ -a_{i_0,j_0} & \text{si } i \in I \ (j_0 \text{ es el único índice tal que } a_{i_0,j_0} = 1) \\ 0 & \text{en otro caso} \end{cases}$$

entonces Lanery muestra que D_{i_0} corresponde a una línea del poliedro. Además, si A_I es una base tal que cada $i_0 \in (F - I)$ verifica la propiedad 3.2, entonces el conjunto $\{D_{i_0} : i_0 \in (F - I)\}$ es una base de la variedad lineal más grande contenida en el poliedro.

Algoritmo para encontrar el marco de un poliedro convexo

Dado un poliedro $\mathcal{P} \doteq [\![AX \leqslant B]\!]_{\mathrm{r}}$, para encontrar el marco que lo representa se procede a realizar los siguientes pasos:

1. Construir la forma estándar $A_0 X = B_0$, $X^E \geqslant 0$, donde $X \in \mathbb{R}^{n+m}$.

2. Aplicar el primer paso de inicialización del método simplex. Si no hay bases factibles, el poliedro es vacío. En caso contrario, se obtiene un sistema $A_1 X = B_1$, $X^E \geqslant 0$ en forma canónica con respecto a la base factible A_{I_1} con $(B_1)^E \geqslant 0$.

3. Mientras exista una variable inicial fuera de la base que satisfaga la condición 3.2, hacer un pivot que pone esta variable dentro de la base, removiendo una variable aumentada de la base.

4. Se obtiene un sistema $A_2 X = B_2$, $X^E \geqslant 0$ en forma canónica con respecto a la base A_{I_2}. Hay que considerar dos casos:

   (a) Si todas las variables iniciales están en la base ($F \subset I_2$), entonces el poliedro no contiene líneas y se encontró un vértice. Recorriendo exhaustivamente todas las bases factibles del sistema se pueden obtener todos los vértices junto con sus rayos adyacentes.

   (b) Si ($F \not\subset I_2$), con lo cual hay $x_{i_1}, \ldots, x_{i_\delta}$ variables iniciales que están fuera de la base y cumplen la condición 3.2. Entonces $D_{i_1}, \ldots, D_{i_\delta}$ es un base de la variedad lineal más grande contenida en el poliedro. El sistema de restricciones:

   $$AX \leqslant B \wedge \Big\langle \forall\, k\ : 0 \leqslant k \leqslant \delta : \Big\langle \textstyle\sum j\ : 0 \leqslant j \leqslant n :\ (d_{i_k})^j x^j = 0 \Big\rangle \Big\rangle$$

   define un nuevo poliedro $\mathcal{P}'$ que es una sección del poliedro inicial $\mathcal{P}$. Aplicando el algoritmo a este nuevo sistema de restricciones, el caso 4a es aplicable ya que el poliedro $\mathcal{P}'$ no contiene líneas y por lo tanto el algoritmo termina.

Simplificación

En general, el marco que se obtiene con el método de Lanery tiene elementos irrelevantes, y análogamente a lo expuesto en la sección 3.2.3, hay que eliminarlos.

En [Lan66] se propone un método de simplificación que elimina todos los miembros irrelevantes del marco generado (cuando el sistema de restricciones del poliedro es conocido). Este método es el dual al ya visto para el sistema de restricciones generado a partir de un marco (83), y se basa en los siguientes resultados:

- Un vértice o un rayo que no satura ninguna desigualdad es irrelevante.

- Un rayo que satura a todas las restricciones es una línea.
- Dados V_i, V_j dos vértices o dos rayos (que no sean líneas), se define una relación de quasi-orden entre ellos de la siguiente manera:

$$V_i \sqsubseteq V_j \equiv V_j \text{ satura a cada desigualdad que es saturada por } V_i$$

  Entonces:

  - Si $V_i \sqsubseteq V_j$ pero $V_i \not\sqsupseteq V_j$, entonces V_i es irrelevante.
  - Si $V_i \sqsubseteq V_j$ y $V_j \sqsubseteq V_i$, uno y sólo uno de los dos elementos es irrelevante y puede ser eliminado.

Aplicando estos resultados al marco antes obtenido, se obtiene un conjunto generador minimal que representa al mismo poliedro con el cual se inició la simplificación. Notar además, que para realizar el procedimiento descripto es necesario conocer ambas representaciones.

Los métodos de no-pivoting

Los métodos de no-pivoting se basan en el algoritmo creado por Motzkin [MRTT53], el cual resuelve iterativamente el problema del cómputo de la representación dual para un cono poliedral[8]. Dado que cualquier poliedro convexo puede ser descompuesto en un cono poliedral y en un poliedro convexo acotado[9] (por el teorema de descomposición propuesto por Motzkin en el año 1936) esta solución puede ser aplicada a cualquier poliedro convexo.

Un bosquejo del método de Motzkin es el siguiente: en cada iteración, una nueva restricción es añadida al cono actual en un *tableau*[10] (que ha sido construido usando un poliedro inicial con ambas representaciones calculadas). Un nuevo cono es creado a partir de una determinada selección de los rayos del cono actual. Dado que esta selección puede abarcar rayos no extremos o redundantes, antes de una nueva iteración se efectúa una simplificación sobre los mismos.

En base al algoritmo anterior han surgido distintas variantes y mejoras del mismo. Un método que es muy nombrado en la literatura consultada es el método de Chernikova [Che64], el cual pertenece a la categoría de no-pivoting, y está basado en el método de Motzkin, pero usa un tableau más chico y mejorado. El mismo resuelve el caso restringido de restricciones mixtas (es decir igualdades y desigualdades), con el añadido de una restricción de no-negatividad de las variables ($x_i \geqslant 0$ para todo $i = 1 \ldots n$).

Una extensión del método de Chernikova es el planteado por Fernández y Quintón en [FQ88], en donde la restricción de no-negatividad es eliminada y se agrega una heurística para ordenar las restricciones del poliedro inicial, y así reducir el tiempo de ejecución del algoritmo.

Le Verge en [Ver92] mejoró aún más la performance del método reduciendo el tiempo que consume el chequeo de rayos redundantes (que hasta entonces era el cuello de botella del algoritmo).

[8] Un cono poliedral es un caso particular de un poliedro, el cual posee un único vértice

[9] Un *poliedro convexo acotado* es un poliedro convexo que no contiene ni rayos ni líneas.

[10] Un *tableau* es una matriz aumentada formada a través del reemplazo de desigualdades por igualdades.

En conclusión, ha habido un gran avance en materia de generación del marco de un poliedro a partir del sistema de restricciones del mismo. La elección de cuál método utilizar tendrá un impacto importante en la eficiencia del algoritmo más general en donde se lo use.

En la actualidad, varias bibliotecas para el manejo de poliedros utilizan en su implementación interna el método de Le Verge como por ejemplo [Tea02, HPR97, Wil93].

3.3 Invariantes lineales en el dominio de poliedros convexos

En esta sección veremos como aplicar el marco de interpretación abstracta desarrollado al dominio de lo poliedros convexos cerrados con la finalidad de obtener invariantes de forma automática. Dentro de esta marco, dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$, nuestro dominio concreto será $\mathsf{Pred}_{\mathcal{L}}$ y la función semántica concreta será $\mathcal{F}_{\mathcal{T},\Theta}$ (propagación hacia adelante) como ya mencionamos en los ejemplos 3.1 y 3.2 (pág. 68). En las próximas secciones definiremos el dominio semántico abstracto, la función semántica abstracta y el operador de widening utilizado.

3.3.1 Abstracción

Procederemos a construir nuestro dominio abstracto como una extensión puntual de la misma forma que lo hicimos con Pred y $\mathsf{Pred}_{\mathcal{L}}$ en el capítulo 1. De esta forma primero estudiaremos el conjunto de poliedros convexos cerrados en $\mathbb{R}^n$ (denominado Poly) como el dominio abstracto para el conjunto de predicados simples Pred.

El poset $(\mathsf{Poly}, \subseteq)$ (poliedros con la inclusión estándar) es un reticulado cuyas operaciones binarias de ínfimo y supremo son la intersección y la envoltura convexa respectivamente. La primera operación puede calcularse utilizando la representación del poliedro como sistema de restricciones y la segunda utilizando la representación como marco; sean dos poliedros $p, q \in \mathsf{Poly}$ cuyas representaciones como sistema de restricciones y marco son

$$\begin{aligned} p &= [\![A_1 X \leqslant B_1]\!]_{\mathrm{r}} = [\![(\mathcal{S}_1, \mathcal{R}_1, \mathcal{D}_1)]\!]_{\mathrm{m}} \\ q &= [\![A_2 X \leqslant B_2]\!]_{\mathrm{r}} = [\![(\mathcal{S}_2, \mathcal{R}_2, \mathcal{D}_2)]\!]_{\mathrm{m}} \ . \end{aligned}$$

Con estas definiciones las operaciones binarias de ínfimo y supremo del reticulado pueden expresarse como:

$$\begin{array}{lcll} p \cap q & \doteq & [\![A_1 X \leqslant B_1 \wedge A_2 X \leqslant B_2]\!]_{\mathrm{r}} & (\sqcap \text{ en } \mathsf{Poly}) \\ p \uplus q & \doteq & [\![(\mathcal{S}_1 \cup \mathcal{S}_2,\ \mathcal{R}_1 \cup \mathcal{R}_2,\ \mathcal{D}_1 \cup \mathcal{D}_2)]\!]_{\mathrm{m}} & (\sqcup \text{ en } \mathsf{Poly}) \ , \end{array}$$

con lo cual, para calcular estas operaciones es necesario conocer ambas representaciones justificando el uso de la doble representación de poliedros ya mencionada.

Para calcular la relación de orden (inclusión) del poset también haremos uso de ambas representaciones. Sean dos poliedros $p, q : \mathsf{Poly}$ entonces se puede demostrar que la inclusión $p \subseteq q$ se cumple si y solo sí los vértices de p están

incluidos en q y los rayos y líneas de p son también rayos (no necesariamente rayos extremos) y líneas de q respectivamente. Para el caso de los vértices solo se debe comprobar que los mismos satisfagan las restricciones de q. Para el caso de las líneas se puede demostrar que una linea pertenece a un poliedro si la misma satura (definición 3.20, pág. 83) sus restricciones. En el caso de los rayos es necesario comprobar que los rayos del poliedro p sean también de los semiespacios definidos por las restricciones de q. Veremos a continuación cuales son las restricciones que se deben comprobar: sea R un rayo de p y $AX \leqslant b$ una restricción de q, utilizando la definición 3.17 (pág. 77) tenemos:

$$\langle \forall \mu, V \ : \mu \in \mathbb{R}^+ \wedge V \in q : \ A(V + \mu R) \leqslant b \rangle$$
$$\equiv \{ \text{ Distributividad, conmutatividad con escalar } \}$$
$$\langle \forall \mu, V \ : \mu \in \mathbb{R}^+ \wedge V \in q : \ AV + \mu AR \leqslant b \rangle$$

Realicemos un análisis separado de los casos $AR \leqslant 0$ y $AR > 0$:

Caso $(AR \leqslant 0)$

$$\langle \forall \mu, V \ : \mu \in \mathbb{R}^+ \wedge V \in q : \ AV + \mu AR \leqslant b \rangle$$
$$\Leftarrow \{ \text{ Por caso } \mu AR \leqslant 0 \ \}$$
$$\langle \forall \mu, V \ : \mu \in \mathbb{R}^+ \wedge V \in q : \ AV + \mu AR - \mu AR \leqslant b \rangle$$
$$\equiv \{ \text{ Aritmética } \}$$
$$\langle \forall \mu, V \ : \mu \in \mathbb{R}^+ \wedge V \in q : \ AV \leqslant b \rangle$$
$$\equiv \{ \ V \text{ pertenece al poliedro entonces satisface la desigualdad } \}$$
$$\mathsf{T}$$

Caso $(AR > 0)$

$$\langle \forall \mu, V \ : \mu \in \mathbb{R}^+ \wedge V \in q : \ AV + \mu AR \leqslant b \rangle$$
$$\equiv \{ \text{ Aritmética } \}$$
$$\langle \forall \mu, V \ : \mu \in \mathbb{R}^+ \wedge V \in q : \ \mu \leqslant (b - AV)/AR \rangle$$
$$\equiv \{ \ \mu \text{ no está acotado } \}$$
$$\mathsf{F}$$

En consecuencia, un rayo R de p es también de q si y solo si $AR \leqslant 0$ para toda desigualdad $AX \leqslant b$ de q. A partir de este análisis podemos deducir la relación de orden en el poset $(\mathsf{Poly}, \subseteq)$: si $p \doteq [\![(\mathcal{S}, \mathcal{R}, \mathcal{D})]\!]_{\mathrm{m}}$ y $q \doteq [\![AX \leqslant B]\!]_{\mathrm{r}}$ entonces

$$p \subseteq q \doteq \langle \forall S, R, D \ : S \in \mathcal{S} \wedge R \in \mathcal{R} \wedge D \in \mathcal{D} : \ AS \leqslant B \wedge AR \leqslant 0 \wedge AD = 0 \rangle \ .$$

El reticulado posee además mínimo $\perp$ (el conjunto vacío $\varnothing$) y máximo $\top$ ($\mathbb{R}^n$).

En general el poset $(\mathsf{Poly}, \subseteq)$ no es cpo local ni co-cpo[11] por lo tanto no es un reticulado completo[12]. Este hecho puede verificarse considerando el limite de

[11] El caso particular de poliedros convexos cerrados en $\mathbb{R}$ (intervalos cerrados) es completo.

[12] Esta incompletitud es la que dificulta la utilización de otro marco de interpretación abstracta basado en la definición de una función de abstracción o por medio de conectores de Galois.

una cadena de poliedros convergente hacia una esfera, la cual no es un poliedro y cualquier poliedro inscripto en ella puede ser refinado agregando un vértice en la misma.

A continuación enumeraremos algunas propiedades de este dominio:

Propiedades 3.27
Sean $p, q, r \in \mathsf{Poly}$ poliedros, entonces

1. $p \cup q \subseteq p \uplus q$
2. $p \cap (q \uplus r) \supseteq (p \cap q) \uplus (p \cap r)$ y su dual,
3. $p \supseteq r \Rightarrow p \cap (q \uplus r) \supseteq (p \cap q) \uplus r$ y su dual.

Cabe mencionar que el reticulado no es modular ni distributivo, aunque cumple las dos últimas propiedades afines por ser reticulado.

Dado un sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ definiremos el dominio abstracto $\mathsf{Poly}_{\mathcal{L}}$ de $\mathsf{Pred}_{\mathcal{L}}$ como la extensión puntual de Poly sobre las locaciones $\mathcal{L}$, esto es

$$\mathsf{Poly}_{\mathcal{L}} \doteq \mathcal{L} \to \mathsf{Poly}$$

y extenderemos puntualmente el máximo, mínimo, las operaciones de intersección y envoltura convexa. Dados $P, Q \in \mathsf{Poly}_{\mathcal{L}}$ definiremos:

$$\begin{array}{rcll} \top.l & \doteq & \top & (\top \text{ en } \mathsf{Poly}_{\mathcal{L}}) \\ \bot.l & \doteq & \bot & (\bot \text{ en } \mathsf{Poly}_{\mathcal{L}}) \\ (P \cap Q).l & \doteq & P.l \cap Q.l & (\sqcap \text{ en } \mathsf{Poly}_{\mathcal{L}}) \\ (P \uplus Q).l & \doteq & P.l \uplus Q.l & (\sqcup \text{ en } \mathsf{Poly}_{\mathcal{L}}) \\ (P \subseteq Q).l & \doteq & \langle \forall l : l \in \mathcal{L} : P.l \subseteq Q.l \rangle & (\text{orden en } \mathsf{Poly}_{\mathcal{L}}) \end{array}$$

En esta sección utilizaremos como espacio de estados indistintamente a la funciones de las variables a los valores $\Sigma \to \mathbb{R}$ o a $\mathbb{R}^n$ con n la cantidad de variables, en la forma que ya fue mencionada en la sección 1.2. Esto último permite ver al conjunto de poliedros como un subconjunto propio de los predicados, es decir $\mathsf{Poly} \subsetneq \mathsf{Pred}$. Para simplificar aún más la exposición, asociaremos directamente un poliedro con su representación sintáctica de sistema de restricciones de la misma forma que abreviamos un predicado con su sintaxis (sección 1.2.1).

De esta forma, con el dominio semántico abstracto ya establecido procederemos definir la función de concretización $\gamma_{\mathcal{L}} : \mathsf{Poly}_{\mathcal{L}} \to \mathsf{Pred}_{\mathcal{L}}$ como extensión puntual de $\gamma : \mathsf{Poly} \to \mathsf{Pred}$. Esto es, si un poliedro $p \in \mathsf{Poly}$ está representado de forma algebraica como $p = [\![AX \leqslant B]\!]_{\mathrm{r}}$ definiremos γ como $\gamma.p \doteq AX \leqslant B$ abreviando la semántica con la sintaxis. Pensando a los poliedro como un subconjunto propio de los predicados también podemos definir a γ como la inyección identidad:

$$\begin{array}{|l} \gamma : \mathsf{Poly} \to \mathsf{Pred} \\ \hline \gamma \doteq \mathsf{Id} \end{array}$$

y con esta definición la función de concretización será:

$$\begin{array}{|l} \gamma_{\mathcal{L}} : \mathsf{Poly}_{\mathcal{L}} \to \mathsf{Pred}_{\mathcal{L}} \\ \hline \gamma_{\mathcal{L}}.P.l \doteq \gamma.(P.l) \end{array}$$

o simplemente $\gamma_{\mathcal{L}} \doteq \mathsf{Id}$ utilizando la extensión puntual de la identidad en $\mathcal{L}$. De esta forma la abstracción utilizada será $(\mathsf{Pred}_{\mathcal{L}}, \mathsf{Poly}_{\mathcal{L}}, \gamma_{\mathcal{L}})$.

3.3.2 Función semántica abstracta

Como ya mencionamos en el ejemplo 3.2 la función semántica concreta será el transformador de propagación hacia adelante $\mathcal{F}_{\mathcal{T},\Theta} : \mathsf{Pred}_{\mathcal{L}} \to \mathsf{Pred}_{\mathcal{L}}$, con lo cual la función semántica abstracta debe estar relacionada con aquella según la definición 3.3 o su lema equivalente 3.4. Para lograrlo vamos a construir un sistema de transiciones abstracto y una jerarquía de transformadores en el dominio abstracto análoga a la definida en el capítulo 1. Dado un sistema de transiciones concreto $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ tendremos

- para cada sentencia $s \in \mathcal{S}$ construiremos un sentencia abstracta $s^\sharp$ las que formarán parte del conjunto $\mathcal{S}^\sharp$,
- una relación de transiciones abstracta $\mathcal{T}^\sharp$ igual que la concreta pero relacionando sentencias abstractas, esto es $\mathcal{T}^\sharp.l.s^\sharp.l' = \mathcal{T}.l.s.l'$,
- un conjunto de estado iniciales abstractos $\Theta^\sharp : \mathsf{Poly}_{\mathcal{L}}$ construido a partir de su contraparte concreta tal que $\Theta \subseteq \gamma_{\mathcal{L}}.\Theta^\sharp$, o simplemente $\Theta \subseteq \Theta^\sharp$ ya que $\gamma_{\mathcal{L}}$ es la función identidad.

Con estos elementos definiremos $\mathsf{TS}^\sharp \doteq (\mathcal{L}, \mathcal{S}^\sharp, \mathcal{T}^\sharp, \Theta^\sharp)$ como el sistema de transiciones abstracto de $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$.

Dado en sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ el transformador $\mathcal{F}_{\mathcal{T},\Theta}$ se definió como $\mathcal{F}_{\mathcal{T},\Theta}.X \doteq \Theta \cup \mathrm{SP}.\mathcal{T}.X$ el cual a su vez está definido en base al transformador sp en Pred. En esta sección haremos lo mismo para definir una función semántica $\mathcal{F}^\sharp_{\mathcal{T}^\sharp,\Theta^\sharp} : \mathsf{Poly}_{\mathcal{L}} \to \mathsf{Poly}_{\mathcal{L}}$ que cumpla con aquella definición y en base a un transformador $\mathrm{sp}^\sharp$ en Poly. Dado que $\mathcal{F}_{\mathcal{T},\Theta}$ en el dominio concreto está definida (secciones 2.3.1 y 1.6.1) como

$$\mathcal{F}_{\mathcal{T},\Theta}.X \doteq \Theta \cup \mathrm{SP}.\mathcal{T}.X$$

con

$$\mathrm{SP}.\mathcal{T}.P.l' \doteq \left\langle \bigcup l, s \ : \mathcal{T}.l.s.l' : \ \mathrm{sp}.s.(P.l) \right\rangle ,$$

de manera análoga definiremos la función semántica abstracta:

$$\mathcal{F}^\sharp_{\mathcal{T}^\sharp,\Theta^\sharp}.X^\sharp \doteq \Theta^\sharp \uplus \mathrm{SP}^\sharp.\mathcal{T}^\sharp.X^\sharp$$

con

$$\mathrm{SP}^\sharp.\mathcal{T}^\sharp.P.l' \doteq \left\langle \biguplus l, s^\sharp \ : \mathcal{T}^\sharp.l.s^\sharp.l' : \ \mathrm{sp}^\sharp.s^\sharp.(P.l) \right\rangle ,$$

reemplazando la cuantificación sobre la unión finita en Pred por la envoltura convexa en Poly y dejando para más adelante las definiciones de las transiciones, sentencias, estados iniciales y transformador $\mathrm{sp}^\sharp$ abstractos.

En este punto veremos una condición suficiente para cumplir el lema 3.4 que define la función semántica abstracta. Dado $P : \mathsf{Poly}_{\mathcal{L}}$ tenemos

$$\mathcal{F}_{\mathcal{T},\Theta}.(\gamma_{\mathcal{L}}.P) \subseteq \gamma_{\mathcal{L}}.(\mathcal{F}^\sharp_{\mathcal{T}^\sharp,\Theta^\sharp}.P)$$

$\equiv$ { Definición de $\mathcal{F}_{\mathcal{T},\Theta}$ y $\mathcal{F}^\sharp_{\mathcal{T}^\sharp,\Theta^\sharp}$. $\gamma_{\mathcal{L}}$ es la identidad. }

$$\Theta \cup \mathrm{SP}.\mathcal{T}.P \subseteq \Theta^\sharp \uplus \mathrm{SP}^\sharp.\mathcal{T}^\sharp.P$$

$\Leftarrow$ { Extensión puntual de propiedad 3.27(1) }

$\Theta \subseteq \Theta^\sharp \wedge \mathrm{SP}.\mathcal{T}.P \subseteq \mathrm{SP}^\sharp.\mathcal{T}^\sharp.P$

$\equiv$ { Definición de SP y SP$^\sharp$, lógica de predicados }

$\Theta \subseteq \Theta^\sharp \wedge$

$$\left\langle \forall\, l' \ : l' \in \mathcal{L} \ : \left\langle \bigcup l, s \ : \mathcal{T}.l.s.l' : \ \mathrm{sp}.s.(P.l) \right\rangle \subseteq \left\langle \bigsqcup l, s^\sharp \ : \mathcal{T}^\sharp.l.s^\sharp.l' : \ \mathrm{sp}^\sharp.s^\sharp.(P.l) \right\rangle \right\rangle$$

$\Leftarrow$ { Propiedad 3.27(1) }

$\Theta \subseteq \Theta^\sharp \wedge$

$\left\langle \forall\, l, s \ : l \in \mathcal{L} \wedge s \in \mathcal{S} : \ \mathrm{sp}.s.(P.l) \subseteq \mathrm{sp}^\sharp.s^\sharp.(P.l) \right\rangle$

$\Leftarrow$ { Lógica de predicados }

$\Theta \subseteq \Theta^\sharp \wedge$

$\left\langle \forall\, p, s \ : p \in \mathsf{Poly} \wedge s \in \mathcal{S} : \ \mathrm{sp}.s.p \subseteq \mathrm{sp}^\sharp.s^\sharp.p \right\rangle$

por lo tanto, si podemos construir sentencias abstractas a partir de sentencias concretas y tenemos un transformador de poliedros $\mathrm{sp}^\sharp.s^\sharp$ tal que para todo poliedro $p : \mathsf{Poly}$ se cumple $\mathrm{sp}.s.p \subseteq \mathrm{sp}^\sharp.s^\sharp.p$ habremos conseguido nuestra función semántica abstracta. A continuación veremos como construir las sentencias abstractas y el transformador de poliedros sobre las mismas.

3.3.3 Sentencias abstractas

Como mencionamos en la sección 1.5.2 para modelar programas podemos restringir el conjunto finito $\mathcal{S}$ a sentencias de la forma $[b];\langle f\rangle$. Esta elección se basa en el hecho que aplicaremos las técnicas de interpretación abstracta sobre programas ejecutables. Supondremos además que las actualizaciones funcionales pueden ser escritas como asignaciones múltiples de la forma $x^{i_1}, \cdots, x^{i_k} := e^{i_1}, \cdots, e^{i_k}$ con $\{x^{i_1}, \cdots, x^{i_k}\} \subseteq \mathsf{Var}$ ya que el conjunto de variables es finito. Teniendo en mente esta forma sintáctica de las sentencias vamos a definir cierto tipo de expresiones y términos que utilizaremos en ellas.

Definición 3.28 (Términos y expresiones lineales)
Caracterizaremos las siguientes categorías sintácticas:

- *Términos aritméticos lineales* de la forma

  $$\text{<aterm>} ::= a_0 + a_1 x^{i_1} + \cdots + a_k x^{i_k}$$

  con $x^{i_j} \in \mathsf{Var}$ y constantes $a_j \in \mathbb{Q}$, $j = 0, \cdots, k$.

- *Términos booleanos lineales* de la forma

  $$\text{<bterm>} ::= \text{<aterm>} \leqslant b \ \mid \ \text{<aterm>} < b \ \mid \ \text{<aterm>} = b$$

  con constantes $b \in \mathbb{Q}$.

- *Expresiones booleanas lineales* de la forma

<bexp> ::= <bterm>
| <bexp> ∧ <bexp>
| <bexp> ∨ <bexp>
| ¬<bexp> □

Además, una suposición $[\mathsf{B}]$ con B una expresión booleana lineal será llamada *suposición lineal* y una asignación de la forma $x^{i_1}, \cdots, x^{i_k} := \mathsf{E}^{i_1}, \cdots, \mathsf{E}^{i_k}$ con E^{i_j} un término aritmético lineal será llamada *asignación lineal*. De forma general, una sentencia s definida de la forma

$$s \doteq [\mathsf{B}]; x^{i_1}, \cdots, x^{i_k} := \mathsf{E}^{i_1}, \cdots, \mathsf{E}^{i_k}$$

será llamada *sentencia lineal* si la suposición y la asignación son lineales. A continuación, procederemos a transformar la suposición y la asignación múltiple de forma separada con el fin de obtener $s^\sharp$.

Transformación de una suposición lineal.

Dada una suposición lineal $[\mathsf{B}]$ procederemos de la siguiente manera:

1. Transformaremos la expresión booleana B a su *forma normal disyuntiva*. La razón de está elección será explicada más adelante (nota en página 95).

2. Cada término booleano lineal en la expresión B que represente una igualdad será reemplazado por una expresión equivalente solo con desigualdades. Esto es, cada término de la forma $<\text{bexp}> = b$ será reemplazado por $<\text{bexp}> \leqslant b \wedge \neg(<\text{bexp}> < b)$.

3. De la forma normal se eliminarán las negaciones en los términos booleanos lineales manteniéndose la equivalencia de la forma original. Esto es, si en la forma normal aparece un términos de la forma

   - $\neg(a_0 + a_1 x^{i_1} + \cdots + a_k x^{i_k} \leqslant b)$ se reemplazará por su equivalente
   $$(-a_0) + (-a_1)x^{i_1} + \cdots + (-a_k)x^{i_k} < -b \ ,$$
   - $\neg(a_0 + a_1 x^{i_1} + \cdots + a_k x^{i_k} < b)$ se reemplazará por su equivalente
   $$(-a_0) + (-a_1)x^{i_1} + \cdots + (-a_k)x^{i_k} \leqslant -b \ .$$

4. Cada símbolo de desigualdad estricta se reemplazará por el de menor o igual. Es decir, cada término booleano de la forma $a_0 + a_1 x^{i_1} + \cdots + a_k x^{i_k} < b$ se reemplazará por $a_0 + a_1 x^{i_1} + \cdots + a_k x^{i_k} \leqslant b$. Con este paso se obtiene un predicado más débil.

De esta manera obtendremos una expresión booleana lineal B' tal que $\mathsf{B} \subseteq \mathsf{B}'$ y escrito en forma normal disyuntiva, sin negaciones y donde todos lo términos son desigualdades no estrictas. Esto es, $\mathsf{B}' \doteq \mathsf{P}_1 \vee \cdots \vee \mathsf{P}_\mathsf{u}$ con cada expresión P_i de la forma $\mathsf{L}_\mathsf{i}^1 \wedge \cdots \wedge \mathsf{L}_\mathsf{i}^{\mathsf{q}_\mathsf{k}}$ y cada término booleano $\mathsf{L}_\mathsf{i}^\mathsf{j}$ de la forma $<\text{bexp}> \leqslant b$. Así, cada expresión booleana P_k será una conjunción de restricciones lineales y por lo tanto representan un poliedro. La representación matricial

como sistema de restricciones lineales de cada poliedro $\mathsf{P_i}$ se obtiene directamente de su expresión conjuntiva, agregando coeficientes nulos $0\,x_k$ cuando la variable x_k no aparezca en cada restricción lineal $\mathsf{L_i^j}$. Su representación como marco se obtiene aplicando el algoritmo descripto en la sección 3.2.4. A partir de esta última se puede realizar la envoltura convexa de los poliedros $\mathsf{P_i}$ reemplazando la disyunciones en la forma normal B' por el operador binario $\uplus$, definiendo así la contraparte abstracta del predicado B:

$$\mathsf{B}^\sharp \;\doteq\; \mathsf{P_1} \uplus \cdots \uplus \mathsf{P_u} \;\;.$$

Puede demostrarse que por la forma en que se obtiene este poliedro, se cumple $\mathsf{B} \subseteq \mathsf{B}^\sharp$, lo cual será necesario al momento de definir $\mathrm{sp}^\sharp.\mathcal{T}^\sharp$. De esta forma, dada una sentencia $s \;\doteq\; [b]$ con $b^\sharp$ representado por el sistema de restricciones lineales $AX \leqslant B$, su contraparte abstracta será definida por

$$[_]^\sharp : \mathsf{Pred} \to \Sigma \to \Sigma \to \mathsf{Bool}$$
$$[b]^\sharp.\sigma.\sigma' \;\doteq\; A(\dot{\sigma}.X) \leqslant B \wedge \sigma = \sigma'$$

donde $\dot{\sigma}.X : \mathbb{R}^n$ es la aplicación puntual de una función $\sigma \in \Sigma$ a cada variable en el vector de variables X. A la construcción $[_]^\sharp$ la denominaremos *suposición abstracta*.

Nota: Cabe también preguntarse si a través de estas transformaciones no hemos debilitado demasiado el predicado original B. Esta pérdida de información en la abstracción se produce al obtener la forma normal disyuntiva (paso 4 en el procedimiento descripto anteriormente) y en el cálculo de la envoltura convexa final. La primer pérdida es necesaria debido a que los poliedros deben ser cerrados. Para el análisis de la segunda hay que señalar que al reemplazar las disyunciones por la envoltura convexa siempre se produce esta pérdida de información (ver propiedad 3.27.1, pág 91). De todas formas, podría suceder que aplicando esta sustitución en la forma normal conjuntiva o incluso antes de construir la forma normal se hubiera logrado obtener una abstracción más fuerte. Si hubiésemos tomado la primera alternativa habríamos obtenido una abstracción más débil ya que en la construcción de la forma normal conjuntiva se utiliza la distributividad de la disyunción respecto a la conjunción la cual, según la propiedad dual 3.27.2, produce términos más débiles en el dominio abstracto. Si tomásemos la segunda alternativa también produciríamos términos más débiles ya que al construir la forma normal disyuntiva se utiliza la distributividad de la conjunción respecto a la disyunción lo cual fortalece el resultado en el dominio abstracto según la misma propiedad (no dual). □

Esta transformación de suposiciones concretas a abstractas funciona únicamente cuando los términos booleanos de la expresión son lineales (la suposición es una sentencia lineal). Si algún término no lo es podemos eliminarlos de la forma normal disyuntiva obteniéndose un predicado más débil que el original. A cuenta de este hecho la estrategia es totalmente válida aunque podemos perder demasiada información. Otra alternativa es estudiar cada uno de estos términos no lineales y reemplazarlos por otros lineales más débiles, por ejemplo, el término $\log.x \geqslant 0$ puede ser reemplazado por $x \geqslant 1$.

Transformación de una asignación múltiple lineal.

La idea general para realizar esta transformación es expresar las sentencias en la forma de transformaciones lineales afines (una transformación lineal mas una traslación constante). Para hacerlo primero se reemplazará la asignación con una equivalente. Dada una asignación lineal $x^{i_1}, \cdots, x^{i_k} := \mathsf{E}^{i_1}, \cdots, \mathsf{E}^{i_k}$ procederemos de la siguiente manera:

1. Se agregarán las variables que no estén en el lado izquierdo de la asignación con expresiones en el lado derecho igual a esa misma variable. De esta manera la sentencia quedará en la forma $x^1, \cdots, x^n := \mathsf{E}^1, \cdots, \mathsf{E}^n$ donde si la variable x^j no esté en $\{x^{i_1}, \cdots, x^{i_k}\}$ entonces $\mathsf{E}^j \doteq x^j$.
2. Cada término aritmético lineal E^{j} se escribirá de la forma $a_0^j + a_1^j x^1 + \cdots + a_n^j x^n$ agregando los sumando correspondientes a las variables que no estén en la expresión con coeficiente a_i^j iguales a cero.

De esta manera la asignación puede representarse como una transformación lineal afín cuya forma matricial es $TX + T_0$ con la matriz cuadrada $T : \mathbb{Q}^{n \times n}$ que representa la transformación lineal definida como $T \doteq [\, a_i^j \mid 1 \leqslant i \leqslant n \wedge 1 \leqslant j \leqslant n \,]$, el vector columna $T_0 : \mathbb{Q}^n$ que representa la traslación definida como $T_0 \doteq [\, a_0^j \mid 1 \leqslant j \leqslant n \,]$ y el vector columna de variables $X \doteq [\, x^j \mid 1 \leqslant j \leqslant n \,]$. De forma esquemática esta transformación puede escribirse como:

$$TX + T_0 \doteq \begin{pmatrix} a_1^1 & a_2^1 & \dots & a_n^1 \\ a_1^2 & a_2^2 & \dots & a_n^2 \\ \vdots & \vdots & \ddots & \\ a_1^n & a_2^n & \dots & a_n^n \end{pmatrix} \begin{pmatrix} x^1 \\ x^2 \\ \vdots \\ x^n \end{pmatrix} + \begin{pmatrix} a_0^1 \\ a_0^2 \\ \vdots \\ a_0^n \end{pmatrix}$$

$$= \begin{pmatrix} a_0^1 + a_1^1 x^1 + a_2^1 x^2 + \dots + a_n^1 x^n \\ a_0^2 + a_1^2 x^1 + a_2^2 x^2 + \dots + a_n^2 x^n \\ \vdots \\ a_0^n + a_1^n x^1 + a_2^n x^2 + \dots + a_n^n x^n \end{pmatrix}$$

$$= \begin{pmatrix} \mathsf{E}^1 \\ \mathsf{E}^2 \\ \vdots \\ \mathsf{E}^{\mathsf{n}} \end{pmatrix}$$

Notar que vista la sentencia de asignación s como una actualización funcional $\langle f \rangle$ con $f : \Sigma \to \Sigma$, la transformación afín es una forma de representar esta misma función en el ámbito de los poliedros. Si f es representada por la transformación $TX + T_0$ entonces podemos definir la abstracción de la sentencia s como $s^\sharp \doteq \langle f \rangle^\sharp$ con

$$\begin{array}{|l} \langle _ \rangle^\sharp : \mathsf{Fun} \to \Sigma \to \Sigma \to \mathsf{Bool} \\ \hline \langle f \rangle^\sharp.\sigma.\sigma' \doteq T(\dot{\sigma}.X) + T_0 = \dot{\sigma}'.X \end{array}$$

A esta construcción la denominaremos *actualización funcional abstracta*.

Esta forma para representar las asignaciones en el dominio abstracto sirve cuando el lado derecho de las mismas son términos aritméticos lineales de las variables. Si la asignación a una variable no posee esta forma se aproximará asumiendo que la variable puede obtener cualquier valor después de la asignación. Esto se logra eliminando la fila de la transformación $TX + T_0$ que corresponde a dicha variable. Así por ejemplo si la asignación en cuestión es $x^1, \cdots, x^n := \mathsf{E}^1, \cdots, \mathsf{E}^n$ con las expresiones $\mathsf{E}^{k_1}, \cdots, \mathsf{E}^{k_p}$ no lineales, se puede construir la transformación $TX + T_0$ con $T : \mathbb{Q}^{(n-p)\times n}$ y $T_0 : \mathbb{Q}^{n-p}$ donde

$$\begin{array}{lcl} T & \doteq & [\,a_i^j \mid 1 \leqslant i \leqslant n \wedge 1 \leqslant j \leqslant n \wedge j \notin \{k_1, \cdots, k_p\}\,] \ , \\ T_0 & \doteq & [\,a_0^j \mid 1 \leqslant j \leqslant n \wedge j \notin \{k_1, \cdots, k_p\}\,] \ y \\ X & \doteq & [\,x^j \mid 1 \leqslant j \leqslant n \wedge j \notin \{k_1, \cdots, k_p\}\,] \ . \end{array}$$

Con esta transformación la definición de la actualización funcional abstracta $\langle f \rangle^\sharp.\sigma.\sigma' \doteq T(\dot{\sigma}.X) + T_0 = \dot{\sigma}'.X$ indicará que los estados relacionados pueden tener cualquier valor en las variables $x^{k_1}, \cdots, x^{k_p}$.

Transformación de sentencias generales.

Teniendo métodos para encontrar las versiones abstractas de las sentencias de suposición y actualización funcional, procederemos a derivar la de sentencias generales de la forma $[b];\langle f \rangle$. Sea las versiones abstractas de estas sentencias individuales $[b]^\sharp$ con su sistema de restricciones $AX \leqslant B$ asociado y $\langle f \rangle^\sharp$ con su transformación afín $TX + T_0$, si definimos la composición secuencial abstracta como composición de relaciones (igual que con las sentencias concretas) tendremos:

$$\begin{array}{ll} & ([b]^\sharp;\langle f \rangle^\sharp).\sigma.\sigma' \\ \equiv & \{ \text{ Definición de composición secuencial } \} \\ & \left\langle \exists\, \sigma'' \ :: \ [b]^\sharp.\sigma.\sigma'' \wedge \langle f \rangle^\sharp.\sigma''.\sigma' \right\rangle \\ \equiv & \{ \text{ Definición de suposición y actualización funcional abstractas } \} \\ & \langle \exists\, \sigma'' \ :: \ A(\dot{\sigma}.X) \leqslant B \wedge \sigma = \sigma'' \wedge T(\dot{\sigma}''.X) + T_0 = \dot{\sigma}'.X \rangle \\ \equiv & \{ \text{ Rango unitario } \} \\ & A(\dot{\sigma}.X) \leqslant B \wedge T(\dot{\sigma}.X) + T_0 = \dot{\sigma}'.X \ . \end{array}$$

Por lo tanto la definición de una sentencia abstracta general será:

$$([b]^\sharp;\langle f \rangle^\sharp).\sigma.\sigma' \doteq A(\dot{\sigma}.X) \leqslant B \wedge T(\dot{\sigma}.X) + T_0 = \dot{\sigma}'.X \ ,$$

lo cual es el resultado de la aplicación de la transformación afín restringiendo el dominio al sistema lineal originado por la sentencia $[b]^\sharp$.

3.3.4 Transformador de poliedros

Desarrollado el método para obtener sentencias abstractas a partir de concretas obtendremos el transformador de strongest postcondition a nivel abstracto. Para sentencias concretas el transformador se definió (sección 1.4, pág. 10) como

$$\mathrm{sp}.s.p.\sigma' \doteq \left\langle \exists\, \sigma \ : p.\sigma : \ s.\sigma.\sigma' \right\rangle \ .$$

Dada la sentencia abstracta $s^\sharp \doteq [b]^\sharp;\langle f\rangle^\sharp$ representada por el sistema de restricciones $AX \leqslant B$ y la transformación afín $TX+T_0$, podemos definir un transformador abstracto $\mathrm{sp}^\sharp.s^\sharp : \mathsf{Poly} \to \mathsf{Poly}$ de forma análoga al concreto utilizando una cuantificación existencial:

$$
\begin{aligned}
& \mathrm{sp}^\sharp.s^\sharp.p.\sigma' \\
\doteq\ & \{ \text{ Definición análoga a sp } \} \\
& \big\langle \exists\, \sigma \ : p.\sigma : \ s^\sharp.\sigma.\sigma' \big\rangle \\
=\ & \{ \text{ Definición de } s^\sharp \ \} \\
& \big\langle \exists\, \sigma \ : p.\sigma : \ ([b]^\sharp;\langle f\rangle^\sharp).\sigma.\sigma' \big\rangle \\
=\ & \{ \text{ Definición de } [b]^\sharp;\langle f\rangle^\sharp \ \} \\
& \langle \exists\, \sigma \ : p.\sigma : \ A(\dot\sigma.X) \leqslant B \wedge T(\dot\sigma.X) + T_0 = \dot\sigma'.X \rangle \\
=\ & \{ \ p \text{ es un poliedro definido por el sistema } CX \leqslant D \ \} \\
& \langle \exists\, \sigma \ : C(\dot\sigma.X) \leqslant D : \ A(\dot\sigma.X) \leqslant B \wedge T(\dot\sigma.X) + T_0 = \dot\sigma'.X \rangle
\end{aligned}
$$

En consecuencia podemos definir $\mathrm{sp}^\sharp.s^\sharp$ como

$$\mathrm{sp}^\sharp.s^\sharp.p.\sigma' \ \doteq\ \big\langle \exists\, \sigma \ : C(\dot\sigma.X) \leqslant D : \ A(\dot\sigma.X) \leqslant B \wedge T(\dot\sigma.X) + T_0 = \dot\sigma'.X \big\rangle \ .$$

Esta expresión puede ser simplificada mediante un reemplazo de la variables de cuantificación σ (que denota funciones de las variables a los valores) por las variables de programa $x^1, \cdots, x^n$ y la variable σ' por las variables primadas $x'^1, \cdots, x'^n$:

$$\mathrm{sp}^\sharp.s^\sharp.p.X' \ \doteq\ \big\langle \exists\, X \ : CX \leqslant D : \ AX \leqslant B \wedge TX + T_0 = X' \big\rangle \ ,$$

donde $X \doteq [\, x^i \mid 0 \leqslant i \leqslant n \,]$ es el vector columna de variables del programa y $X' \doteq [\, x'^i \mid 0 \leqslant i \leqslant n \,]$ es el vector de variables primadas.

Queda por corroborar que el resultado de aplicar este transformador da como resultado un poliedro. La última definición muestra que el transformador strongest postcondition abstracto es denotado con una cuantificación existencial sobre un sistema de restricciones lineales entre variables de programa y variables primadas:

$$
\begin{aligned}
& \mathrm{sp}^\sharp.s^\sharp.p.X' \ \doteq \\
& \quad \langle \exists\, X \ :: \ CX \leqslant D \wedge AX \leqslant B \wedge TX + T_0 \leqslant X' \wedge TX + T_0 \geqslant X' \rangle \ .
\end{aligned}
$$

Aplicando la técnica de eliminación de cuantificación existencial vista en la sección 3.2.3 puede obtenerse un sistema de restricciones lineales sobre las variables libres (primadas) mediante la aplicación sucesiva de la operación proyección sobre las variables cuantificadas (sin primar). Por lo tanto no solo se demuestra que el resultado del transformador es un poliedro, si no que además tenemos un algoritmo para obtener el sistema de restricciones que lo denota. Resumiendo, si eliminamos las variables no primadas del sistema de restricciones en $\mathbb{R}^{2n}$

$$AX \leqslant B \wedge TX + T_0 \leqslant X' \wedge TX + T_0 \geqslant X'$$

proyectándolo sobre el espacio de las variables primadas, obtenemos un nuevo sistema en $\mathbb{R}^n$ de la forma $A'X' \leqslant B'$. Este último representa el poliedro resultado de la transformación:

$$\mathrm{sp}^\sharp.s^\sharp.[\![AX \leqslant B]\!]_{\mathrm{r}} \ \doteq\ [\![A'X' \leqslant B']\!]_{\mathrm{r}} \ .$$

Cabe agregar que el método de eliminación existencial puede simplificarse si la transformación lineal T es invertible, de manera análoga a lo que sucede en el dominio concreto (sección 1.4.1, propiedades 1.4, pág. 12):

$$
\begin{aligned}
& \mathrm{sp}^\sharp.s^\sharp.p.X' \\
= & \ \{ \text{ Definición } \} \\
& \left\langle \exists X \ : CX \leqslant D : \ AX \leqslant B \wedge TX + T_0 = X' \right\rangle \\
= & \ \{ \text{ Álgebra } \} \\
& \left\langle \exists X \ : CX \leqslant D : \ AX \leqslant B \wedge X = T^{-1}(X' - T_0) \right\rangle \\
= & \ \{ \text{ Intercambio rango y término } \} \\
& \left\langle \exists X \ : X = T^{-1}(X' - T_0) : \ CX \leqslant D \wedge AX \leqslant B \right\rangle \\
= & \ \{ \text{ Rango unitario } \} \\
& C(T^{-1}(X' - T_0)) \leqslant D \wedge A(T^{-1}(X' - T_0)) \leqslant B
\end{aligned}
$$

Por lo tanto, para este caso, el resultado del transformador puede ser obtenido como:

$$
\mathrm{sp}^\sharp.s^\sharp.[\![AX \leqslant B]\!]_{\mathrm{r}} \ \doteq \ [\![C(T^{-1}(X' - T_0)) \leqslant D \wedge A(T^{-1}(X' - T_0)) \leqslant B]\!]_{\mathrm{r}} \ .
$$

3.3.5 Sistemas de transiciones abstractos

Como ya mencionamos en la sección 3.3.2, dado un sistema de transiciones concreto $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ podemos derivar uno abstracto $\mathsf{TS}^\sharp \ \doteq \ (\mathcal{L}, \mathcal{S}^\sharp, \mathcal{T}^\sharp, \Theta^\sharp)$ de la siguiente manera:

- Para cada sentencia $s \in \mathcal{S}$ construimos un sentencia abstracta $s^\sharp$ en la forma descripta en la sección 3.3.3, las que formarán parte del conjunto $\mathcal{S}^\sharp$.

- La relación de transiciones abstracta $\mathcal{T}^\sharp$ será igual que la concreta pero relacionando sentencias abstractas, esto es $\mathcal{T}^\sharp.l.s^\sharp.l' = \mathcal{T}.l.s.l'$.

- El conjunto de estado iniciales abstractos $\Theta^\sharp : \mathsf{Poly}_\mathcal{L}$ será construido abstrayendo cada formula en $\Theta : \mathsf{Pred}_\mathcal{L}$ de la misma forma que lo hicimos con las expresiones booleanas en las sentencias de suposición (pág. 94). De esta forma aseguramos que se cumpla $\Theta \subseteq \Theta^\sharp$.

Con este sistema en mente y a partir de la definición del transformador de poliedros de la sección anterior, podemos definir la función semántica abstracta ya esbozada en la sección 3.3.2:

$$
\mathcal{F}^\sharp_{\mathcal{T}^\sharp, \Theta^\sharp}.X^\sharp \ \doteq \ \Theta^\sharp \uplus \mathrm{SP}^\sharp.\mathcal{T}^\sharp.X^\sharp
$$

con

$$
\mathrm{SP}^\sharp.\mathcal{T}^\sharp.P.l' \ \doteq \ \left\langle \biguplus l, s^\sharp \ : \mathcal{T}^\sharp.l.s^\sharp.l' : \ \mathrm{sp}^\sharp.s^\sharp.(P.l) \right\rangle \ .
$$

Finalizada la definición de la función semántica abstracta, procederemos a completar el marco de interpretación abstracta definiendo un operador de widening apropiado en el dominio de poliedros.

3.3.6 Widening

El operador de widening utilizado en este trabajo es el propuesto originalmente por Cousot y Halbwachs [CH78] denominado *widening estándar* por su uso frecuente en distintas herramientas de verificación.

Dados dos poliedros p, q : Poly el resultado de aplicar el operador de widening sobre los mismos será el poliedro convexo definido por las restricciones de p que incluyen al poliedro q. Esto es, dado $p \doteq [\![A^1X \leqslant b^1 \wedge \cdots \wedge A^mX \leqslant b^m]\!]_{\mathrm{r}}$ (A^i son las filas de la matriz A) y $q \doteq [\![(\mathcal{S}, \mathcal{R}, \mathcal{D})]\!]_{\mathrm{m}}$, definimos el operador de widening

$$p \nabla q \doteq [\![\langle \forall i : 1 \leqslant i \leqslant m \wedge q \subseteq [\![A^iX \leqslant b^i]\!]_{\mathrm{r}} : A^iX \leqslant b^i \rangle]\!]_{\mathrm{r}} .$$

Una manera equivalente de definir el operador es haciendo explícito el subconjunto de filas $J \subseteq [1..m]$ de la matriz A del poliedro $p \doteq [\![AX \leqslant B]\!]_{\mathrm{r}}$ definido como:

$$j \in J \equiv q \subseteq [\![A^jX \leqslant b^j]\!]_{\mathrm{r}}$$

o incluyendo implementación de inclusión expuesta en la sección 3.3.1:

$$\begin{array}{l} j \in J \equiv \langle \forall S, R, D : S \in \mathcal{S} \wedge R \in \mathcal{R} \wedge D \in \mathcal{D} : \\ \qquad\qquad A^jS \leqslant b^j \wedge A^jR \leqslant 0 \wedge A^jD = 0 \rangle . \end{array}$$

Con este conjunto de índices así definido tenemos

$$p \nabla q \doteq [\![\langle \forall j : j \in J : A^jX \leqslant b^j \rangle]\!]_{\mathrm{r}} .$$

Notar que para implementar este operador de widening hace falta conocer ambas representaciones de los poliedros lo cual justifica también la necesidad de la doble representación.

Este operador de widening cumple las condiciones en la definición 3.6 (pág. 71). La condición $p \subseteq p \nabla q$ se cumple ya que el conjunto de restricciones que representan a $p \nabla q$ son un subconjunto de las que representan a p. La condición $q \subseteq p \nabla q$ también se cumple ya que q está incluido en las restricciones de $p \nabla q$. Además se cumple el segundo criterio en la definición ya que para toda cadena de poliedros $x_0 \subseteq x_1 \subseteq \cdots \subseteq x_n \subseteq \cdots$ la cadena $x'_0 = x_0, x'_1 = x'_o \nabla x_1, \cdots, x'_n = x'_{n-1} \nabla x_n, \cdots$ es finita ya que en cada paso el número de restricciones que representa x'_n es finito y es un subconjunto de las de x'_{n-1}.

El operador de widening así definido trabaja solo sobre poliedros en Poly. Para utilizarlo en este trabajo hace falta extenderlo puntualmente al dominio abstracto $\mathsf{Poly}_{\mathcal{L}}$:

$$\begin{array}{|l} _ \nabla_{\mathcal{L}} _ : \mathsf{Poly}_{\mathcal{L}} \to \mathsf{Poly}_{\mathcal{L}} \to \mathsf{Poly}_{\mathcal{L}} \\ \hline (P \nabla_{\mathcal{L}} Q).l \doteq P.l \nabla Q.l \end{array}$$

Como mencionamos al principio de esta sección, el operador descripto es clásico en la literatura y es utilizado en muchas herramientas de verificación. Cabe agregar que otros operadores han sido estudiados y propuestos en [BHRZ03, Hal06].

3.3.7 Invariantes lineales

Dado un sistema de transiciones concreto $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ y recapitulando lo desarrollado en esta sección, los elementos necesarios obtenidos para implementar el marco de interpretación abstracta de la sección 3.1 (pág. 67) son

- la abstracción $(\mathsf{Pred}_{\mathcal{L}}, \mathsf{Poly}_{\mathcal{L}}, \gamma_{\mathcal{L}})$ (sección 3.3.1),
- la función semántica abstracta $\mathcal{F}^{\sharp}_{\mathcal{T}^{\sharp},\Theta^{\sharp}} : \mathsf{Poly}_{\mathcal{L}} \to \mathsf{Poly}_{\mathcal{L}}$ (sección 3.3.2) definida como
$$\mathcal{F}^{\sharp}_{\mathcal{T}^{\sharp},\Theta^{\sharp}}.X \ \doteq\ \Theta^{\sharp} \uplus \mathrm{SP}^{\sharp}.\mathcal{T}^{\sharp}.X$$
con
$$\mathrm{SP}^{\sharp}.\mathcal{T}^{\sharp}.P.l' \ \doteq\ \left\langle \biguplus l, s^{\sharp} \ : \mathcal{T}^{\sharp}.l.s^{\sharp}.l' : \ \mathrm{sp}^{\sharp}.s^{\sharp}.(P.l) \right\rangle \ ,$$
- el operador de widening $\nabla_{\mathcal{L}}$ (sección anterior).

Con estos elementos podemos implementar el algoritmo 3.3 (pág. 74) que genera un invariante a partir de la función semántica abstracta:

Programa 3.4 Generación de invariantes

aproximar_$\mu.\mathcal{F}_{\mathcal{T},\Theta}$ $(\mathcal{F}^{\sharp}_{\mathcal{T}^{\sharp},\Theta^{\sharp}} : \mathsf{Poly}_{\mathcal{L}} \to \mathsf{Poly}_{\mathcal{L}},$
$\nabla_{\mathcal{L}}: \mathsf{Poly}_{\mathcal{L}} \to \mathsf{Poly}_{\mathcal{L}} \to \mathsf{Poly}_{\mathcal{L}},$
$\mathrm{M} : \mathbb{N}) : \mathsf{Poly}_{\mathcal{L}}$

$k := 0;$
$G^{\sharp}_k := \bot^{\sharp};$
<u>**do**</u> $k < M \wedge \neg(\mathcal{F}^{\sharp}_{\mathcal{T}^{\sharp},\Theta^{\sharp}}.G^{\sharp}_k \sqsubseteq G^{\sharp}_k)$
  $G^{\sharp}_k := \mathcal{F}^{\sharp}_{\mathcal{T}^{\sharp},\Theta^{\sharp}}.G^{\sharp}_k;$
  $k := k + 1$
<u>**od**</u>;
<u>**do**</u> $\neg(G^{\sharp}_k \ \nabla_{\mathcal{L}}\ \mathcal{F}^{\sharp}_{\mathcal{T}^{\sharp},\Theta^{\sharp}}.G^{\sharp}_k \ \sqsubseteq\ G^{\sharp}_k)$
  $G^{\sharp}_k := G^{\sharp}_k \ \nabla_{\mathcal{L}}\ \mathcal{F}^{\sharp}_{\mathcal{T}^{\sharp},\Theta^{\sharp}}.G^{\sharp}_k$
<u>**od**</u>;
return $G^{\sharp}_k$

El algoritmo devuelve un invariante del sistema de transiciones concreto $\gamma.(\mu\mathcal{F}_{\mathcal{T},\Theta})^{\sharp} : \mathsf{Pred}_{\mathcal{L}}$. A partir del mismo y dada una propiedad P se puede verificar su invariancia comprobando la implicación $\gamma.(\mu\mathcal{F}_{\mathcal{T},\Theta})^{\sharp} \sqsubseteq P$ (propiedad 2.11(3), pág. 44). Este uso directo del invariante generado tiene dos deficiencias. Primero, la implicación anterior puede no ser decidible para predicados P arbitrarios. Esta limitación se afronta restringiendo los predicados de forma tal que existan procedimientos de decisión para la verificación de la implicación. Como ejemplo, se puede restringir P a que sea representable de manera exacta con poliedros convexos para que la implicación sea decidible (como se demostró en la sección 3.3.1).

La segunda deficiencia es que en la práctica, el algoritmo suele encontrar invariantes demasiado débiles para probar propiedades arbitrarias. Esto es, el algoritmo encuentra el invariante $\gamma.(\mu\mathcal{F}_{\mathcal{T},\Theta})^\sharp : \mathsf{Pred}_\mathcal{L}$ tal que $\mu.\mathcal{F}_{\mathcal{T},\Theta} \subseteq \gamma.(\mu\mathcal{F}_{\mathcal{T},\Theta})^\sharp$ (teorema 3.8), pero no siempre es válida la condición $\gamma.(\mu\mathcal{F}_{\mathcal{T},\Theta})^\sharp \subseteq P$ cuando P es un invariante ya que el invariante encontrado es solo una sobre aproximación de $\mu\mathcal{F}_{\mathcal{T},\Theta}$.

Por esta razón y como se mostrará en el capítulo 5, en este trabajo solo se utiliza el método expuesto para encontrar un invariante del sistema que luego será empleado a fin de acelerar el proceso de convergencia de la demostración de k-invariancia en el programa 2.3 (pág. 62). Esta utilización de los invariantes lineales generados como aceleradores de convergencia es también empleada en [BBM97].

3.3.8 Implementación

La implementación del algoritmo de generación de invariantes lineales fue realizada como parte del trabajo de tesis final de la Lic. Natalia Beatriz Bidart [Bid07]. Este desarrollo fue incluido en el prototipo de software que describiremos en el capítulo 5.

El algoritmo fue escrito en el lenguaje ML, en su implementación *Standard ML of New Jersey* [Sta06], utilizando la biblioteca externa *NewPolka* [Jea05] para la manipulación de poliedros.

Capítulo 4

Implementación de regiones críticas condicionales

En capítulos anteriores desarrollamos una formalización de los sistemas de transiciones junto con diferentes técnicas para comprobar propiedades sobre los mismos. La principal motivación de este trabajo es la aplicación de estas técnicas para la optimización de construcciones concurrentes utilizadas en los lenguajes de programación. Se entiende a estas construcciones como estructuras sintácticas de alto nivel que permiten una abstracción del hardware o de primitivas del lenguaje para resolver de forma simple problemas de concurrencia, como por ejemplo regiones críticas condicionales y monitores. Por lo general, debido al alto nivel de estas construcciones, su implementación automática por medio de compiladores o generadores de código suele ser poco eficiente.

En este mismo sentido, la técnica de semáforos binarios divididos [Dij79] puede ser usada para implementar regiones críticas condicionales. Dada una especificación de un problema de esta clase, SBD brinda tanto los programas que lo implementan como los invariantes que aseguran su corrección. Aplicando la técnica a casos particulares se encuentran programas que admiten ciertas simplificaciones que mejoran la eficiencia. Este trabajo se concentra en el desarrollo de procedimientos automáticos para obtenerlas. Nos enfocamos particularmente en la eliminación de guardas en las sentencias condicionales finales de los programas, pero el método permite su generalización a otros tipos de simplificaciones.

El procedimiento consiste en representar de manera abstracta una implementación SBD de regiones críticas condicionales por medio de sistemas de transiciones (capítulo 1) y hacer una búsqueda de nuevos invariantes (capítulo 2) que avalen la corrección de las simplificaciones. La técnica principalmente utilizada para ello es *propagación hacia atrás* (sección 2.3.2, pág.53). Como vimos, este método permite constatar la invariancia de predicados sobre los sistemas de transiciones y posee la ventaja de que la cadena tendiente al punto fijo es usualmente finita.

Una ventaja adicional que posee la técnica de propagación hacia atrás es que los predicados intermedios son fórmulas libres de cuantificadores como se mencionó en el capítulo 2. De esta manera, la obtención de los mismos puede hacerse de forma mecánica con la ayuda de chequeadores de validez de fórmulas de primer orden con teorías subyacentes denominados *SMT solvers*, como por

ejemplo CVC3 [BT07, CVC08] y Yices [yic08].

Desafortunadamente las fórmulas producidas en el proceso de cálculo de punto fijo son generalmente grandes ya que en cada paso del cálculo iterativo se produce un incremento en el tamaño de las mismas. Con el fin de aliviar este fenómeno, se desarrollaron métodos de simplificación de las fórmulas intermedias, implementados con los probadores de teoremas CVC3 e Isabelle/HOL [Pau94].

En la actualidad las herramientas denominadas en forma genérica "probadores de teoremas" han demostrado una gran madurez, siendo utilizadas de manera exitosa en la verificación de software. Nuestro trabajo tiene un objetivo diferente: pretende atacar el problema de eficiencia antes mencionado utilizando estas herramientas.

La propuesta de este trabajo fue implementada en un prototipo de software escrito en el lenguaje de programación ML utilizando demostradores de teoremas externos para probar las implicaciones envueltas en el cálculo de punto fijo y simplificar fórmulas lógicas. El procedimiento fue probado sobre diferentes ejemplos clásicos de programación concurrente.

La técnica propuesta es incompleta ya que puede no detectar guardas cuya eliminación es correcta. Sin embargo en la mayoría de los ejemplos de la literatura se detectaron todas las simplificaciones posibles. Cabe remarcar que esta incompletitud no es un problema a la hora de obtener programas correctos: las implementaciones de regiones críticas condicionales producidas por la técnica SBD sin estas simplificaciones son correctas aunque menos eficientes. En este sentido, el objetivo de aplicación del método automático propuesto se dirige a implementaciones de compiladores o generadores de código para aumentar la eficiencia de construcciones concurrentes de alto nivel.

El trabajo ha sido presentado en [BB07a, BB07b]. Aquí profundizaremos estas publicaciones en detalle. En las secciones 4.1, 4.2 y 4.3 se plantea el problema con sus antecedentes. En las secciones 4.4, 4.5, 4.6 y 4.7 se presenta el método propuesto con su implementación. Por último en las secciones 4.8 y 4.9 se muestran los resultados con sus conclusiones y los trabajos futuros propuestos.

4.1 Regiones críticas condicionales

Una gran cantidad de problemas concurrentes pueden ser resueltos usando los conceptos de espera condicional y exclusión mutua. Por ejemplo en [And89] se presenta un método relativamente general para resolver problemas de sincronización usando *espera condicional* $\langle \textbf{await}.\mathsf{B} \mapsto \mathsf{S} \rangle$ el cual sirve para resolver ambos problemas iniciales (espera condicional y exclusión mutua). De manera general, las esperas condicionales funcionan bloqueando el proceso que ejecuta el programa hasta que la condición B sea verdadera y en este caso el programa S se ejecuta en exclusión mutua.

Las construcciones de espera pueden aparecer en cualquier punto de los programas causando que las variables en la condiciones B y en los programas S sean compartidas por los distintos procesos que las ejecutan. Esta libertad del contexto en donde aparecen las variables produce algunas dificultades al momento de implementarlas. La forma clásica para implementar esta construcción es a través de bucles *busy wait* en cada guarda de las distintas esperas condicionales. El principal problema reside en que se deben chequear todas las guardas que

aparecen en las distintas esperas de los procesos. Además, cuando se realiza esta comprobación, se deben bloquear todos los procesos que usen las variables de las guardas para que no interfieran en el resultado de esta evaluación. Es importante agregar que en muchos contextos de ejecución concurrente, es común que existan guardas que no sea necesario evaluar, dejando como tarea al programador la eliminación de las mismas para una mejora en la eficiencia del sistema.

A partir de esta dificultad en la implementación de las esperas condicionales, diversas construcciones más simples de sincronización han sido propuestas (como por ejemplo semáforos). Estos mecanismos son menos abstractos que la espera condicional y pueden usarse para implementar esta construcción. Esta perdida de abstracción introduce nuevas necesidades de prueba en el proceso de desarrollo de los programas, incrementando su complejidad y por consiguiente generando nuevas posibilidades de cometer errores. La presencia de errores sutiles, tanto de corrección (safety) como de progreso (en particular la posibilidad de deadlocks o livelocks) ha sido desafortunadamente más la regla que la excepción. Toda propuesta de nuevas herramientas de sincronización ha sido siempre una solución de compromiso entre facilidad de uso y posibilidad de reducir las penalidades en eficiencia.

En este sentido, las *regiones críticas condicionales* fueron propuestas por C.A.R. Hoare [Hoa72] y Brinch Hansen [Han72] como una construcción más cercana a la espera condicional y por lo tanto más fácil de usar correctamente. Las mismas se presentan con una notación estructurada para especificar sincronización, donde se hace necesario declarar explícitamente las variables compartidas sobre las cuales se operará en exclusión mutua y que podrán aparecer en las condiciones. La construcción de alto nivel para denotar regiones críticas condicionales requiere que cualquier variable compartida v sea declarada como:

$$\underline{\textbf{resource}}\ rcc\ (v)$$

De esta manera se obliga a que la variable v pueda ser usada solo dentro de una región declarada como:

$$\underline{\textbf{region}}\ \ rcc\ \square\ \mathsf{B} \mapsto \mathsf{S}\ \underline{\textbf{end}}$$

Esta construcción significa que mientras el programa S esta siendo ejecutado, ningún proceso puede acceder a la variable v y que la guarda B gobierna el acceso a esta región crítica. Además la ejecución de S debe realizarse en exclusión mutua.

Una manera de implementar regiones críticas condicionales con construcciones más simples fue propuesta por E.Dijkstra [Dij79] retomando ideas previas de C.A.R. Hoare [Hoa74]. La técnica fue denominada *semáforos binarios divididos* (SBD) y utiliza una serie de semáforos binarios para asegurar la exclusión mutua entre las regiones. Esta metodología brinda tanto los programas (con semáforos binarios) que implementan las regiones críticas condicionales como los invariantes iniciales que aseguran su corrección.

Debido a la generalidad de la metodología, los programas resultantes suelen ser poco eficientes; como veremos, en estos programas aparecen guardas en sentencias condicionales que son falsas en cualquier contexto de ejecución posible. Es por esto que, para mejorar la eficiencia, la técnica incluye la eliminación,

mediante demostraciones de corrección en forma manual, de estos chequeos innecesarios en los puntos en los cuales puede deducirse formalmente que una condición será falsa. Dado que las secciones críticas suelen ser pequeñas pero son invocadas numerosas veces, estos pequeños ahorros pueden representar incrementos drásticos en la eficiencia de los programas.

El desarrollo en los últimos años de los demostradores (semi)automáticos de teoremas brinda un nuevo contexto sobre el cual una parte interesante de estas simplificaciones puede hacerse de manera mecánica, abriendo la posibilidad de reducir significativamente las penalidades en eficiencia en las de construcciones de alto nivel para la construcción de programas concurrentes. Esta nueva alternativa resulta totalmente relevante en el contexto actual ya que la necesidad de crear programas con estas características no es nueva aunque ha tomado particular interés en los últimos tiempos. La popularidad de arquitecturas paralelas, con el advenimiento de los procesadores multicore, ha renovado la necesidad de elaboración de técnicas y construcciones concurrentes para aprovechar la características de estas arquitecturas. El problema surge debido a que actualmente estas metodologías están lejos de cumplir el desafío. Conocidas catástrofes económicas fueron producidas por errores en programas concurrentes [Gib94]. Problemas al intentar aprovechar estas arquitecturas para la actualización del sistemas operativos SunOS están descriptos en [Cre05]. Son también destacables casos como el desarrollo del kernel de Ptolemy II [EJL+02] donde se descubrieron errores mucho tiempo después al probarlos en estas arquitecturas, aún cuando el desarrollo estuvo altamente testeado [Lee06].

Uno de los problemas centrales de las ciencias de la computación es la de proveer construcciones concurrentes de alto nivel sin penalidades en la eficiencia. Al momento de atacar este problema, las construcciones concurrentes de alto nivel propuestas, como regiones críticas condicionales y monitores con señalamiento implícito, no han tenido buena aceptación debido a sus problemas de eficiencia. La madurez que en la actualidad han obtenido los probadores de teoremas los hace una herramienta interesante a la hora de resolver estas cuestiones. Nuestro trabajo intenta mostrar esta posibilidad tomando una construcción de alto nivel (regiones críticas condicionales) y mejorándola incrementando su eficiencia gracias al uso de este tipo de herramientas.

A continuación explicaremos la técnica SBD junto con un ejemplo de simplificación del programa resultante hecho en forma manual con el fin de visualizar el problema que se intenta resolver.

4.2 Técnica SBD

Los semáforos fueron desarrollados por E. W. Dijkstra [Dijnd] como una herramienta de programación para sincronizar el acceso a recursos compartidos en un ambiente concurrente. Esta construcción es presentada en la forma de un tipo abstracto de datos consistente de una variable especial denominada semáforo, asociada a un valor dentro de un intervalo de enteros no negativos comenzando de cero, junto con las operaciones P y V sobre la misma. De manera general, dado un semáforo s, al ejecutarse una operación P.s dentro de una componente concurrente, esta detiene su ejecución si el semáforo posee el valor cero. En el caso que sea mayor, se decrementa este valor de manera atómica y la componente continúa su ejecución. Por otro lado, si en una componente se ejecuta

una operación V.s, el semáforo incrementa en forma atómica su valor, siempre que no se haya llegado al límite superior de su rango, en cuyo caso la operación es neutra sobre el semáforo. Esto permite que si el semáforo hubiera tenido el valor nulo, las posibles componentes bloqueadas en operaciones P.s continúen su ejecución. Los semáforos que admiten cualquier valor no negativo son denominados *semáforos generales* y los restringidos a los valores 0 y 1 son llamados semáforos binarios. En estos últimos, la operación V incrementa el semáforo solo si tiene el valor cero. Una descripción más detallada de esta construcción puede encontrarse en [And91, cap. 4].

Los semáforos binarios pueden asegurar de manera muy simple exclusión mutua y por lo tanto son buenos candidatos para implementar regiones críticas. Una manera particular de usar los semáforos binarios provee un método para implementar regiones críticas condicionales. Describiremos aquí estas ideas brevemente, remitiendo a la literatura para una presentación más completa [Dij79, Dij80b, And99, Sch97, Hoo86, MvdS89, Hoo90, BB07a].

Un conjunto $\mathsf{SBD} \doteq \{s_0, \ldots, s_n\}$ de semáforos binarios se denominará *semáforo binario dividido*[1] si en cualquier momento de la ejecución del programa a lo sumo uno de ellos toma el valor 1. Esto es equivalente a requerir la invariancia de la siguiente propiedad:

$$0 \leqslant \langle \sum i \ : 0 \leqslant i \leqslant n : \ s_i \rangle \leqslant 1 \ .$$

Toda ejecución de una región crítica comenzará entonces dinámicamente con una operación P sobre alguno de los elementos del SBD y terminará con un V sobre un elemento del mismo conjunto (no necesariamente el mismo). El invariante garantiza entonces exclusión mutua entre estas dos operaciones.

Además de la exclusión mutua, los semáforos en SBD satisfacen la siguiente *regla del dominó* [MvdS89]: si la ejecución de una región crítica termina con una operación V sobre un semáforo s, entonces la próxima operación P deberá ocurrir sobre el mismo semáforo s. Esto permite asumir la precondición de cualquier operación V como poscondición de su correspondiente operación P. Esta regla puede formularse en términos axiomáticos como la invariancia global del siguiente predicado:

$$\varphi_{SBD} \ \doteq \ \langle \forall s \ : s \in \mathsf{SBD} : \ s = 0 \ \vee \ I_s \rangle \tag{4.1}$$

donde I_s es un predicado que se cumple antes de todo comando V.s y después de su correspondiente P.s.

De forma resumida, la técnica para implementar regiones críticas condicionales consiste en asociar cada elemento del conjunto SBD con una condición de la región crítica. Será también necesario agregar un semáforo "neutral" para el caso en el que ninguna condición sea verdadera. Luego, toda región crítica estará dinámicamente prefijada por un P asociado con su precondición. Además, debe tenerse cierto cuidado para introducir suficientes operaciones V con el fin de asegurar progreso.

A continuación ilustraremos el método de forma general solo con dos regiones condicionales para facilitar la exposición.

[1] El nombre del conjunto es el mismo que el que utilizamos para denominar la técnica. Esta ambigüedad será resuelta según el contexto en el que se presente.

Sean dos programas S_0, S_1 asumiendo como precondiciones respectivamente B_0 y B_1 y estados iniciales de las variables θ, nuestro objetivo será implementar las regiones críticas condicionales

resource *rcc* (v) Precondición: θ	
SCC_0: **region** *rcc* ▯ $\mathsf{B}_0 \mapsto \mathsf{S}_0$ **end**	SCC_1: **region** *rcc* ▯ $\mathsf{B}_1 \mapsto \mathsf{S}_1$ **end**

Supongamos además que las regiones críticas deben preservar cierto invariante I. Usaremos entonces un SBD compuesto por dos semáforos $\mathsf{SBD} \doteq \{s_0, s_1\}$ uno para cada condición y otro semáforo m para cuando ninguna de las dos se satisfaga. Dos contadores b_0, b_1 serán necesarios para contar la cantidad de procesos comprometidos respectivamente con la ejecución de $\mathrm{P}.s_0, \mathrm{P}.s_1$ y poder así asegurar la ausencia de deadlocks.

El siguiente invariante caracteriza la solución basada en SBD:

$$\begin{array}{rl}\varphi_{_{SBD}} \doteq & (s_0 = 0 \;\vee\; (\mathsf{B}_0 \wedge b_0 > 0)) \wedge \\ & (s_1 = 0 \;\vee\; (\mathsf{B}_1 \wedge b_1 > 0)) \wedge \\ & (m = 0 \;\vee\; ((\neg\mathsf{B}_0 \;\vee\; b_0 = 0) \wedge (\neg\mathsf{B}_1 \;\vee\; b_1 = 0))) \\ & \;\wedge\; b_0 \geqslant 0 \wedge b_1 \geqslant 0 \wedge I\end{array}$$

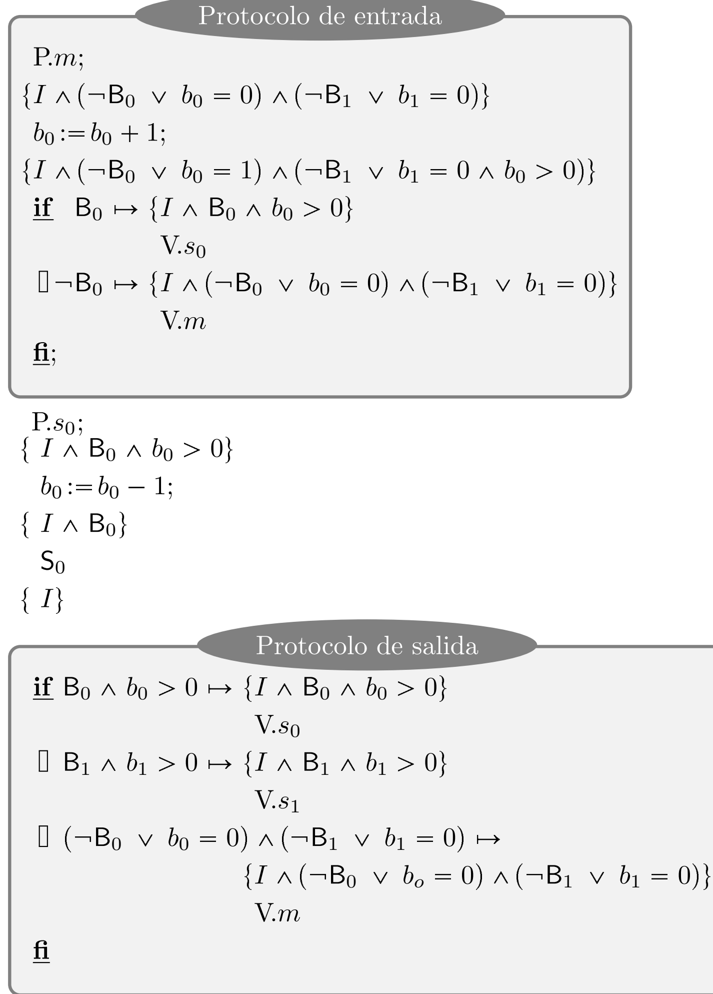
donde los primeros tres términos conjuntivos representan, cada uno, la parte del invariante $s = 0 \;\vee\; I_s$ en la ecuación 4.1 que le corresponde a un semáforo entre la ejecución de las operaciones P-V. Los dos términos siguientes representan las restricciones de los contadores (deben ser mayores e iguales a cero) y el último es el invariante propio del problema.

El programa 4.1 (pág. 109) muestra la implementación obtenida por la técnica SBD, junto con su anotación completa, para la primer región crítica condicional. En el programa se ha señalado el código correspondiente a la parte del protocolo de entrada y salida de la región crítica. La primera se ejecuta solamente si el semáforo neutral m está habilitado. Notar que su poscondición indica que no puede haber procesos esperando ingresar a sus regiones si sus condiciones correspondientes son verdaderas (los contadores b_0 y b_1 iguales a cero indican que no hay procesos esperando). A continuación se verifica la condición B_0. Si esta condición es falsa no es necesario liberar procesos ejecutando S_0 bloqueados en s_0. Además, por la anterior poscondición, si hay procesos en s_1 $(b_1 > 0)$ su condición no se cumple y tampoco es necesario desbloquearlos. Por lo tanto, la única alternativa en este caso es habilitar procesos nuevos, permitiendo el progreso, ejecutando $\mathrm{V}.m$. En el caso que la guarda sea verdadera se habilita el semáforo (con el comando $\mathrm{V}.s_0$) para permitir reasumir alguna ejecución de S_0.

Después de la ejecución del protocolo de entrada, el proceso queda esperando en el semáforo de su condición (semáforo s_0) hasta que sea despertado. Se puede notar que aquí hay una redundancia de esperas en semáforos, no tenida en cuenta en el protocolo de entrada: si se cumple B_0 no puede haber otros procesos esperando en s_0 (por poscondición de $\mathrm{P}.m$), por lo cual el mismo proceso que verificó la guarda será el liberado. En este sentido, más adelante veremos como simplificar el programa.

El protocolo de salida se ejecuta después de acceder a la región crítica. Aquí se verifica si hay procesos en espera cuya condición sea verdadera. En

Programa 4.1 Componente SCC_0 generada por SBD



estos casos se despierta el proceso correspondiente. Notar que esta elección se hace de manera no determinista lo que permite agregar prioridades a la implementación final. En el caso que no se cumpla ninguna de estas condiciones, se libera el semáforo m asegurando progreso.

Como ya se mencionó, esta solución puede simplificarse de manera general distribuyendo ambas asignaciones a b_0 (también puede distribuirse el incremento dado que b_0 no aparece en B_0), y simplificando luego el cuerpo en la primer alternativa (eliminando las operaciones P-V y luego el incremento seguido del decremento de b_0). La simplificación mejora algo la eficiencia al evitar una salida innecesaria de una región crítica. La nueva solución, utilizada en este trabajo, se muestra en el programa 4.2 (pág. 110).

El método SBD es aplicable a problemas con una cantidad arbitraria de regiones críticas condicionales. De manera general, la técnica SBD toma como entrada un conjunto $\{\mathsf{S}_0, \cdots, \mathsf{S}_{m-1}\}$ de programas a ejecutarse en exclusión mu-

Programa 4.2 Componente SCC_0 generada por SBD

$$
\begin{array}{l}
\mathrm{P}.m;\\
\{I \wedge (\neg \mathsf{B}_0 \vee b_0 = 0) \wedge (\neg \mathsf{B}_1 \vee b_1 = 0)\}\\
\underline{\mathbf{if}}\ \ \mathsf{B}_0 \mapsto \{I \wedge \mathsf{B}_0 \wedge b_0 = 0 \wedge (\neg \mathsf{B}_1 \vee b_1 = 0)\}\\
\qquad\qquad \mathbf{skip}\\
\ \square \neg \mathsf{B}_0 \mapsto \{I \wedge \neg \mathsf{B}_0 \wedge (\neg \mathsf{B}_1 \vee b_1 = 0)\}\\
\qquad\qquad b_0 := b_0 + 1\ ;\\
\qquad\qquad \{I \wedge (\neg \mathsf{B}_0 \vee b_0 = 0) \wedge (\neg \mathsf{B}_1 \vee b_1 = 0)\}\\
\qquad\qquad \mathrm{V}.m;\\
\qquad\qquad \mathrm{P}.s_0;\\
\qquad\qquad \{I \wedge \mathsf{B}_0 \wedge b_0 > 0\}\\
\qquad\qquad b_0 := b_0 - 1\\
\underline{\mathbf{fi}};\\
\{I \wedge \mathsf{B}_0\}\\
\mathsf{S}_0\\
\{I\}\\
\underline{\mathbf{if}}\ \mathsf{B}_0 \wedge b_0 > 0 \mapsto \{I \wedge \mathsf{B}_0 \wedge b_0 > 0\}\\
\qquad\qquad\qquad \mathrm{V}.s_0\\
\ \square\ \mathsf{B}_1 \wedge b_1 > 0 \mapsto \{I \wedge \mathsf{B}_1 \wedge b_1 > 0\}\\
\qquad\qquad\qquad \mathrm{V}.s_1\\
\ \square\ (\neg \mathsf{B}_0 \vee b_0 = 0) \wedge (\neg \mathsf{B}_1 \vee b_1 = 0) \mapsto\\
\qquad\qquad\qquad \{I \wedge (\neg \mathsf{B}_0 \vee b_o = 0) \wedge (\neg \mathsf{B}_1 \vee b_1 = 0)\}\\
\qquad\qquad\qquad \mathrm{V}.m\\
\underline{\mathbf{fi}}
\end{array}
$$

tua, junto con sus condiciones asociadas $\{\mathsf{B}_0, \cdots, \mathsf{B}_{m-1}\}$, un conjunto de estados iniciales θ y un invariante global I, donde m es el número de regiones críticas condicionales que se desea implementar. Como resultado de su aplicación la técnica genera de manera automática una serie de m programas $\mathsf{SCC}_0, \cdots, \mathsf{SCC}_{m-1}$ que implementan las regiones críticas condicionales con semáforos. El esquema de un programa SCC_i se muestra en 4.3 (pág. 111). En el mismo se utilizan m semáforos $s_0, \cdots, s_{m-1}$ mas uno neutral s_m, junto con m contadores $b_0, \cdots, b_{m-1}$, siendo el invariante de la implementación SBD:

$$
\begin{aligned}
\varphi_{SBD} \doteq\ & \langle \forall i\ : 0 \leqslant i < m :\ s_i = 0\ \vee (\mathsf{B}_i \wedge b_i > 0) \rangle \\
& \wedge (s_m = 0\ \vee \langle \forall j\ : 0 \leqslant j < m :\ \neg \mathsf{B}_j\ \vee\ b_j = 0 \rangle) \\
& \wedge \langle \forall j\ : 0 \leqslant j < m :\ b_j \geqslant 0 \rangle \wedge I\ .
\end{aligned} \tag{4.2}
$$

El tipo de simplificaciones realizadas en esta última implementación son aplicables a cualquier problema de regiones críticas condicionales. Muchos problemas específicos admiten otras simplificaciones adicionales que dependen de las particularidades de los mismos. Como ya se mencionó, el objetivo de este trabajo es realizar estas simplificaciones de manera automática. Con el fin de

Programa 4.3 Componente SCC_i generada por SBD

$$
\begin{array}{l}
\quad \mathrm{P}.s_m; \\
\quad \underline{\textbf{if}}\ \ \mathsf{B}_i \mapsto \textbf{skip} \\
\quad \square \neg\mathsf{B}_i \mapsto b_i := b_i + 1\ ; \\
\qquad\qquad\quad \mathrm{V}.s_m; \\
\qquad\qquad\quad \mathrm{P}.s_i; \\
\qquad\qquad\quad b_i := b_i - 1 \\
\quad \underline{\textbf{fi}}; \\
\{I \wedge \mathsf{B}_i\} \\
\mathsf{S}_i \\
\{I\} \\
\quad \underline{\textbf{if}}\ \mathsf{B}_0 \wedge b_0 > 0 \mapsto \mathrm{V}.s_0 \\
\quad \square\ \mathsf{B}_1 \wedge b_1 > 0 \mapsto \mathrm{V}.s_1 \\
\quad \vdots \\
\quad \square\ \mathsf{B}_{m-1} \wedge b_{m-1} > 0 \mapsto \mathrm{V}.s_{m-1} \\
\quad \square\ \langle \forall j\ : 0 \leqslant j < m :\ \neg\mathsf{B}_j\ \vee\ b_j = 0 \rangle \mapsto \mathrm{V}.s_m \\
\quad \underline{\textbf{fi}}
\end{array}
$$

mostrar las características de las mismas, presentamos a continuación el siguiente ejemplo tomado de [Hoo90].

Ejemplo 4.1 (Productor/Consumidor en buffer acotado)
Consideremos el problema clásico del *Productor/Consumidor* comunicándose a través de un buffer acodado. El problema contiene dos clases de procesos: los procesos de tipo *productor* producen ciertos datos que serán enviados a través de un buffer acotado a procesos de tipo *consumidor* para su utilización. Se necesita sincronización para evitar escribir en un buffer lleno o leer de un buffer vacío y esto determina las correspondientes guardas. Además la lectura y escritura del buffer debe estar dentro de una región crítica. A continuación mostraremos los programas de estos procesos junto con sus regiones críticas condicionales:

<table>
<tr><td colspan="2">$\underline{\textbf{resource}}\ \textit{prod_cons}\ (p : \textbf{int}, d : \textbf{int}, \textit{buf} : \textbf{array}[0, N)\ \textbf{of}\ T)$
$\theta\ \doteq\ p = 0 \wedge d = 0 \wedge N > 0$</td></tr>
<tr><td>produce.x;
Prod: $\underline{\textbf{region}}\ \textit{prod_cons}$
$\square\, p - d < N \mapsto$
$\textit{buf}.(p\ \textit{mod}\ N) := x;$
$p := p + 1$
$\underline{\textbf{end}}$</td><td>Cons: $\underline{\textbf{region}}\ \textit{prod_cons}$
$\square\, p - d > 0 \mapsto$
$y := \textit{buf}.(d\ \textit{mod}\ N);$
$d := d + 1;$
$\underline{\textbf{end}};$
consume.y</td></tr>
</table>

La constante N es el tamaño del buffer, p es el contador de elementos producidos y d es el de elementos consumidos.

Con el fin de eliminar detalles irrelevantes, nos vamos a concentrar solo en la sincronización, dejando de lado el acceso al buffer, el cual puede ser agregado a

la solución final. Además, ya que las condiciones de las regiones críticas solo hacen referencia a la diferencia entre p y d, realizaremos un cambio de coordenadas a una sola variable n igual a $p - d$. Esta variable denotará la cantidad de elementos que se encuentran en el buffer. Haciendo estas modificaciones, podemos encontrar el siguiente problema equivalente:

resource *prod_cons* (n : **int**) $\theta \doteq n = 0 \land N > 0$	
Prod: **region** *prod_cons* $\square\, n < N \mapsto n := n + 1$ **end**	Cons: **region** *prod_cons* $\square\, n > 0 \mapsto n := n - 1$ **end**

De esta manera, gracias a que las regiones atómicas consisten de solo un comando, puede encontrarse fácilmente el siguiente invariante global:

$$I \doteq 0 \leqslant n \leqslant N \ .$$

Con esta especificación del problema implementaremos las regiones críticas condicionales utilizando la técnica SBD con tres semáforos, uno para cada condición (s y t) más uno neutral (m) que asegure exclusión mutua cuando las condiciones no se cumplan o no hayan procesos esperando entrar en las regiones. Además agregaremos los dos contadores de procesos b y c. El invariante que caracteriza la implementación SBD será entonces:

$$\begin{aligned} \varphi_{SBD} \doteq \ & (s = 0 \ \lor (n < N \land b > 0)) \land \\ & (t = 0 \ \lor (n > 0 \land c > 0)) \land \\ & (m = 0 \lor ((n = N \ \lor \ b = 0) \land (n = 0 \ \lor \ c = 0))) \\ & \land 0 \leqslant n \leqslant N \land b \geqslant 0 \land c \geqslant 0 \ . \end{aligned}$$

Notar que las condiciones fueron simplificadas usando el invariante global I.

A continuación se muestran los programas generados por la técnica:

Programa 4.4 Productor/Consumidor

Prod:	Cons:
P.m;	P.m;
if $n < N \mapsto$ **skip**	**if** $n > 0 \mapsto$ **skip**
$\square\ n = N \mapsto b := b + 1$; V.$m$;	$\square\ n = 0 \mapsto c := c + 1$; V.$m$;
P.s ; $b := b - 1$	P.t ; $c := c - 1$
fi;	**fi**;
$n := n + 1$;	$n := n - 1$;
if $n < N \land b > 0 \mapsto$ V.s	**if** $n < N \land b > 0 \mapsto$ V.s
$\square\ n > 0 \land c > 0 \ \mapsto$ V.t	$\square\ n > 0 \land c > 0 \ \mapsto$ V.t
$\square\ (n = N \ \lor \ b = 0) \land$	$\square\ (n = N \ \lor \ b = 0) \land$
$(n = 0 \ \lor \ c = 0) \mapsto$ V.m	$(n = 0 \ \lor \ c = 0) \mapsto$ V.m
fi	**fi**

La solución puede ser simplificada eliminando varias condiciones a la salida de las secciones críticas (**if** final), dado que puede asegurarse que nunca serán

satisfactibles. Evitar el chequeo de estas guardas tendrá un impacto importante en la eficiencia de este programa, dado que en general las secciones críticas no son extensas pero son ejecutadas muchas veces.

La primera observación sobre estos programas es que después de haberse producido un elemento el buffer el mismo no puede quedar vacío. Utilizando esta idea mostraremos la corrección de la condición de prueba marcada con signo de pregunta:

Programa 4.5 Productor

Prod:

P.m;
<u>if</u> $n < N \mapsto$ **skip**
$\square$ $n = N \mapsto b{:=}b+1$; V.m ; P.s ; $b{:=}b-1$
<u>fi</u>;
$\{n < N\}$
$n{:=}n+1$;
$\{n > 0?\}$
<u>if</u> $n < N \wedge b > 0 \mapsto$ V.s
$\square$ $n > 0 \wedge c > 0 \mapsto$ V.t
$\square$ $(n = N \vee b = 0) \wedge (n = 0 \vee c = 0) \mapsto$ V.m
<u>fi</u>

Este predicado puede ser probado fácilmente ya que wp.$(n{:=}n{+}1).(n > 0)$ es igual a $n \geqslant 0$ lo cual es implicado por el invariante global I. El análisis simétrico puede ser aplicado al programa consumidor obteniéndose la precondición $n < N$ de su **<u>if</u>** final (después del decremento de la variable n). Además, con estas anotaciones, las guardas de las sentencias pueden ser simplificadas:

Programa 4.6 Productor/Consumidor

Prod:	Cons:
P.m;	P.m;
<u>if</u> $n < N \mapsto$ **skip**	**<u>if</u>** $n > 0 \mapsto$ **skip**
$\square$ $n = N \mapsto b{:=}b+1$; V.m ;	$\square$ $n = 0 \mapsto c{:=}c+1$; V.m ;
P.s ; $b{:=}b-1$	P.t ; $c{:=}c-1$
<u>fi</u>;	**<u>fi</u>**;
$\{n < N\}$	$\{n > 0\}$
$n{:=}n+1$;	$n{:=}n-1$;
$\{n > 0\}$	$\{n < N\}$
<u>if</u> $n < N \wedge b > 0 \mapsto$ V.s	**<u>if</u>** $b > 0 \mapsto$ V.s
$\square$ $c > 0 \mapsto$ V.t	$\square$ $n > 0 \wedge c > 0 \mapsto$ V.t
$\square$ $(n = N \vee b = 0) \wedge c = 0 \mapsto$ V.m	$\square$ $b = 0 \wedge (n = 0 \vee c = 0) \mapsto$ V.m
<u>fi</u>	**<u>fi</u>**

Observando el comportamiento operacional de los programas, resulta poco probable que un productor libere (mediante una operación V) a otro productor, lo cual implicaría la posible eliminación de la primer guarda del **<u>if</u>** final en este programa. Esta sospecha en el comportamiento del sistema surge de la observación que la asignación $n{:=}n+1$ no puede volver verdadero al predicado

$n < N$. Para justificar este hecho, es necesario un razonamiento más complejo, pero una característica del método aquí presentado es que si se sospecha sobre la validez de cierta propiedad solo tenemos que intentar probarla. En este sentido, el procedimiento que se describe a continuación será totalmente automatizado.

Para eliminar esta guarda necesitaremos probar la validez de la condición de prueba (negación de la guarda) señalada con signo de pregunta:

Programa 4.7 Productor

Prod:

P.m;
<u>if</u> $n < N \mapsto$ **skip**
$\square$ $n = N \mapsto b := b + 1$; V.m ; P.s ; $b := b - 1$
<u>fi</u>;
$\{n < N\}$
$n := n + 1$;
$\{n > 0\}\{n = N \lor b = 0?\}$
<u>if</u> $n < N \land b > 0 \mapsto$ V.s
$\square$ $c > 0 \mapsto$ V.t
$\square$ $(n = N \lor b = 0) \land c = 0 \mapsto$ V.m
<u>fi</u>

Calculando el transformador de weakest precondition sobre este predicado obtenemos que $n + 1 = N \lor b = 0$ debe cumplirse después del primer **<u>if</u>**. Sobre su primera rama esta condición se cumple debido a que es implicada por la componente del invariante I_m correspondiente al semáforo m. Aplicando el transformador sobre la segunda se obtiene el predicado $n + 1 = N \lor b = 1$ como poscondición para P.s. El método clásico para su verificación consiste en fortalecer de manera guiada el invariante I_s, de forma tal que implique la condición de prueba. De esta forma propondremos fortalecer la parte del invariante correspondiente al semáforo s:

$$s = 0 \lor (n + 1 = N \land b > 0) \ .$$

Este nuevo invariante fortalecido requiere que todo V.s tenga como precondición $n + 1 = N \land b > 0$, en particular dentro del programa consumidor. A continuación se detallan los programas con sus anotaciones y esta última condición de prueba:

Programa 4.8 Productor/Consumidor

Prod:

$P.m\ \{n = N \vee b = 0\};$
$\underline{\textbf{if}}\ \ n < N \mapsto \{b = 0\}\ \textbf{skip}$
$\square\ \ n = N \mapsto b := b + 1;\ V.m\ ;$
$\qquad\qquad P.s\ ;\ b := b - 1$
$\qquad\qquad \{n + 1 = N\}$
$\underline{\textbf{fi}};$
$\{n < N\}\ \{n + 1 = N \vee b = 0\}$
$n := n + 1;$
$\{n > 0\}\ \{n = N \vee b = 0\}$
$\underline{\textbf{if}}\ \ n < N \wedge b > 0 \mapsto V.s$
$\square\ \ c > 0 \mapsto V.t$
$\square\ \ (n = N \vee b = 0) \wedge c = 0 \mapsto V.m$
$\underline{\textbf{fi}}$

Cons:

$P.m;$
$\underline{\textbf{if}}\ \ n > 0 \mapsto \textbf{skip}$
$\square\ \ n = 0 \mapsto c := c + 1;\ V.m\ ;$
$\qquad\qquad P.t\ ;\ c := c - 1$
$\underline{\textbf{fi}};$
$\{n > 0\}$
$n := n - 1;$
$\{n < N\}$
$\underline{\textbf{if}}\ \ b > 0 \mapsto \{n + 1 = N?\}\ V.s$
$\square\ \ n > 0 \wedge c > 0 \mapsto V.t$
$\square\ \ b = 0 \wedge (n = 0 \vee c = 0) \mapsto V.m$
$\underline{\textbf{fi}}$

La misma requiere como precondiciones válidas a $n{+}1 = N \vee b = 0$ antes de su $\underline{\textbf{if}}$ final y $n = N \vee b = 0$ antes del decremento $n := n{-}1$. En el $\underline{\textbf{if}}$ precedente, el invariante asociado al semáforo m asegura su validez, pero en la operación P.t su nueva poscondición deberá cumplir $n = N \vee b = 0$. La estrategia usual en estos casos es fortalecer la parte del invariante correspondiente al semáforo t con esta condición:

$$t = 0 \vee (c > 0 \wedge n > 0 \wedge (n = N \vee b = 0)) \ .$$

Este invariante genera nuevas condiciones de prueba con respecto a las operaciones V.t, o sea deben tener como precondición válida el fortalecimiento $n = N \vee b = 0$. La anotaciones en el anterior programa productor muestran que este requerimiento adicional es alcanzado mientras que en el consumidor queda la obligación de prueba señalada:

Programa 4.9 Consumidor

Cons:

$P.m\ \{n = N \vee b = 0\};$
$\underline{\textbf{if}}\ \ n > 0 \mapsto \textbf{skip}$
$\square\ \ n = 0 \mapsto c := c + 1;\ V.m\ ;$
$\qquad\qquad P.t\ ;\ c := c - 1$
$\qquad\qquad \{n = N \vee b = 0\}$
$\underline{\textbf{fi}};$
$\{n > 0\}\ \{n = N \vee b = 0\}$
$n := n - 1;$
$\{n < N\}\ \{n + 1 = N \vee b = 0\}$
$\underline{\textbf{if}}\ \ b > 0 \mapsto \{n + 1 = N\}\ V.s$
$\square\ \ n > 0 \wedge c > 0 \mapsto \{n = N \vee b = 0?\}\ V.t$
$\square\ \ b = 0 \wedge (n = 0 \vee c = 0) \mapsto V.m$
$\underline{\textbf{fi}}$

Esta última condición no puede ser verificada a menos que fortalezcamos nuevamente el invariante del sistema. La forma de este fortalecimiento puede

intuirse observando la naturaleza simétrica del problema e intentando eliminar también la segunda guarda del último comando **if** del consumidor (un consumidor no necesita liberar a otro consumidor). Esto trae como consecuencia la necesidad de fortalecer aún más el invariante y procediendo de manera similar puede encontrarse el invariante final:

$$\begin{aligned}
&\varphi_{SBD} \doteq \\
&\quad (s = 0 \ \vee b > 0 \wedge n < N \wedge (n + 1 = N \ \vee \ 1 = b) \wedge (c \leqslant 1 \ \vee \ n = 0)) \wedge \\
&\quad (t = 0 \ \vee c > 0 \wedge n > 0 \wedge (1 = n \ \vee \ 1 = c) \wedge (b \leqslant 1 \ \vee \ n = N)) \wedge \\
&\quad (m = 0 \vee ((n = N \ \vee \ b = 0) \wedge (n = 0 \ \vee \ c = 0)))
\end{aligned}$$

resultando una implementación SBD con las guardas eliminadas:

Programa 4.10 Productor/Consumidor

```
Prod:                                  Cons:
  P.m;                                   P.m;
  if  n < N ↦ skip                       if  n > 0 ↦ skip
  ▯  n = N ↦ b:=b+1; V.m ;               ▯  n = 0 ↦ c:=c+1; V.m ;
              P.s ; b:=b−1                           P.t ; c:=c−1
  fi;                                    fi;
  n:=n+1;                                n:=n−1;
  if  c > 0 ↦ V.t                        if  b > 0 ↦ V.s
  ▯  c = 0 ↦ V.m                         ▯  b = 0 ↦ V.m
  fi                                     fi
```

Nótese que dos guardas han sido completamente eliminadas y las otras simplificadas, pasando de evaluar ocho desigualdades a sólo dos. El método propuesto en este trabajo sólo realiza la eliminación de las guardas, no las simplificaciones. Sin embargo, dado que los invariantes ya están calculados el proceso de simplificación de guardas es elemental.

Cabe adelantar que mediante el método propuesto en este trabajo se obtiene un invariante más débil y no es necesario eliminar ambas guardas al mismo tiempo. Esto es debido a la naturaleza de la técnica donde utilizamos propagación hacia atrás lo cual envuelve el cálculo del mayor punto fijo mostrado en la sección 2.3.2 (pág. 53).

A partir del desarrollo del ejemplo puede vislumbrarse la metodología propuesta en este trabajo: partiendo del invariante inicial, se elige una guarda a eliminar y se propaga, mediante el transformador wp, la negación de la guarda en cuestión. El proceso fortalece aquel invariante de manera gradual (obteniéndose invariantes intermedios) hasta deducir la validez de la negación de la guarda. Como se mostrará más adelante, el fortalecimiento gradual del invariante es análogo al método iterativo de propagación hacia atrás expuesto en la sección 2.3.2 (pág. 53), donde la propiedad a demostrar es justamente la negación de la guarda.

El problema de verificación y eliminación de guardas finales superfluas en implementaciones de regiones críticas condicionales no es nuevo. A continuación se detallan algunos antecedentes de nuestro trabajo.

4.3 Antecedentes

La idea general de la técnica SBD fue descubierta por C.A.R. Hoare en [Hoa74] como una forma de implementar monitores con semáforos. En ese artículo la misma no es recomendada por razones de eficiencia, aconsejándose implementar los monitores directamente en el hardware o en los sistemas operativos. La técnica fue después sistematizada por E.W. Dijkstra en [Dij79]. Allí además se demuestran de forma manual las mejoras que haremos de manera automática en nuestro trabajo.

Como antecedente de este trabajo, en la literatura se pueden encontrar dos trabajos que intentan aplicar técnicas asercionales de carácter mecánico para el análisis y mejoramiento de eficiencia. En [Cla79] se muestra un método para encontrar invariantes de regiones críticas condicionales de manera automática. El mismo consiste en el cálculo del menor punto fijo del transformador de propagación hacia adelante, utilizando solamente la técnica de abstracción por poliedros convexos presentada en el capítulo 3. El método presentado solo puede aplicarse a sistemas de regiones críticas condicionales con una cantidad de procesos en ejecución fijo y pequeño. En la publicación se muestra la aplicación del método sobre el problema de lectores y escritores con solo dos lectores y un escritor. Nuestro trabajo resuelve el mismo tipo de problema para una cantidad no acotada de procesos haciendo una abstracción de las colas de procesos como se verá en la sección siguiente. La complejidad de este modelo de concurrencia hace que el uso exclusivo del método de Cousot no alcance para obtener invariantes lo suficientemente fuertes como para eliminar los chequeos innecesarios. En nuestro trabajo se intentó usar esta técnica pero los invariantes obtenidos fueron más débiles que los invariantes que provee la técnica SBD (ecuación 4.2) lo cual no permite resolver el problemas de eliminación de guardas finales.

Otro trabajo, aún más cercano al nuestro, es presentado en [Sch76]. El mismo muestra un método para implementar regiones críticas condicionales de manera eficiente, disminuyendo la cantidad de chequeos innecesarios para una cantidad no acotada de procesos. Dado un conjunto de regiones críticas condicionales:

$$\mathsf{RCC}_i : \underline{\textbf{region}}\ rcc\ \square\ \mathsf{B}_i \mapsto \mathsf{S}_i\ \underline{\textbf{end}} \quad \text{con} \quad i = 1, \cdots, m-1\ ,$$

el método construye una relación ea entre pares de regiones críticas condicionales tal que $(\mathsf{RCC}_k, \mathsf{RCC}_j) \in ea$ si la ejecución de S_k puede cambiar de false a true la evaluación de B_j en cualquier elemento del espacio de estados (no solamente los estados alcanzables). Notar que esta fue la intuición inicial para eliminar la primera guarda en el ejemplo anterior, aunque hubo que demostrarla. A partir de esta relación se implementa un sistema de colas de espera donde un proceso RCC_k libera a un proceso encolado RCC_j si $(\mathsf{RCC}_k, \mathsf{RCC}_j) \in ea$. De esta forma, después de ejecutarse RCC_k solo hay que evaluar las guardas de los procesos bloqueados ejecutando RCC_j siempre que $(\mathsf{RCC}_k, \mathsf{RCC}_j) \in ea$.

Esta simple idea para eliminar chequeos innecesarios no funciona sobre implementaciones SBD. Tomemos como contraejemplo el siguiente sistema de regiones críticas condicionales, el cual es una variación del problema productor/consumidor: al problema del ejemplo 4.1 agreguemos una componente que consuma siempre y cuando el buffer tenga solo un elemento, esto es:

<table>
<tr><td colspan="3"><u>resource</u> p_c (n : int)
$\theta \doteq n = 0 \wedge N > 0 \qquad I \doteq 0 \leqslant n \leqslant N$</td></tr>
<tr><td>Prod: <u>region</u> p_c
$\Box\, n < N \mapsto$
$n := n + 1$
<u>end</u></td><td>Cons1: <u>region</u> p_c
$\Box\, n = 1 \mapsto$
$n := n - 1$
<u>end</u></td><td>Cons: <u>region</u> p_c
$\Box\, n > 0 \mapsto$
$n := n - 1$
<u>end</u></td></tr>
</table>

En este ejemplo la relación ea será:

$$\begin{aligned} ea \doteq \{&(\mathsf{Prod}, \mathsf{Cons}), (\mathsf{Prod}, \mathsf{Cons1}), \\ &(\mathsf{Cons1}, \mathsf{Prod}), (\mathsf{Cons1}, \mathsf{Cons1}), \\ &(\mathsf{Cons}, \mathsf{Prod}), (\mathsf{Cons}, \mathsf{Cons1})\} \ . \end{aligned}$$

Por la misma se puede deducir que un productor no puede liberar a otro productor. Como veremos este resultado no es aplicable a implementaciones SBD. Veamos la siguiente traza de ejecución con un buffer de tamaño dos ($N = 2$) donde se ejecutan cuatro procesos p_1, p_2, p_3, p_4 de tipo Prod, $c1$ de tipo Cons1 y c de tipo Cons, en este orden:

Ejecución	Buffer	En espera
p_1, p_2, p_3, p_4	• •	Prod = $[p_3, p_4]$
$c1$	• •	Prod = $[p_3, p_4]$, Cons1 = $[c1]$
c	_ •	Prod = $[p_3, p_4]$, Cons1 = $[c1]$
$c1$	_ _	Prod = $[p_3, p_4]$, Cons1 = $\varnothing$
p_3	• _	Prod = $[p_4]$, Cons1 = $\varnothing$

En la configuración final el proceso p_4 queda bloqueado lo cual muestra que la solución no asegura el progreso del sistema.

Nuestro método resuelve correctamente este problema ya que no elimina la posibilidad que un productor pueda liberar otro de su misma clase. Visto en el sentido de la publicación [Sch76], nuestro método agrega el par (Prod, Prod) a la relación ea. Además, refina aún más esta relación eliminando el par (Cons1, Cons1) el cual resulta innecesario (un consumidor de tipo Cons1 no necesita liberar a procesos de su misma clase). El resultado de la aplicación de nuestro método a este problema, puede verse al final de este capítulo (sección 4.8.2).

4.4 SBD como sistema de transiciones

En esta sección explicaremos como representar los programas generados por la técnica SBD con sistemas de transiciones. En [MP91] se propone la manera clásica de modelar procesos concurrentes con estas estructuras. La idea general consiste en obtener primero un sistema de transiciones para cada proceso aislado (de la forma ilustrada en el capítulo 1) y a partir de ellos construir su composición paralela como un producto, según la semántica clásica de interleaving utilizada en concurrencia. Al ser este producto finito, la cantidad de procesos que involucra el mismo también debe serlo. Además, para denotar el estado en

un momento de la ejecución, se agrega una variable extra que denota los valores de los contadores de programa de cada proceso. De esta forma, el valor de la variable es una n-upla de elementos en $\mathcal{L}$ (una coordenada por cada proceso).

Un problema que surge al intentar modelar de esta manera el comportamiento de los procesos concurrentes es que en nuestro caso la cantidad de procesos no está acotada. Por más que las cantidad de regiones críticas a implementar sea una constante del problema, los cantidad de procesos que las ejecutan no lo es. Por ejemplo, en el problema Productor/Consumidor solo hay dos regiones críticas a implementar, pero la cantidad de procesos productores y consumidores no se conoce a priori. De esta manera resulta imposible obtener el producto finito de la composición paralela por lo que se hace necesario abstraer de alguna forma el modelo de concurrencia. Establecer una cota para estos procesos (a la manera en [Cla79], sección anterior) resulta artificial, y como veremos, innecesario. En vez de esta alternativa, construiremos un sistema de transiciones diferente que represente acciones atómicas más extensas y más abstractas. Esto es posible gracias a ciertas particularidades de los programas generados por SBD:

Exclusión mutua: Como se mencionó en la sección 4.2 todos los programas comienzan su ejecución con una operación P y terminan con una operación V. Además, todos las sentencias entre ellas son ejecutadas en exclusión mutua. Esto es una característica de las implementaciones brindadas por el método: SBD asegura exclusión mutua entre cualquier par de operaciones P y V, i.e. a los sumo un semáforo se encuentra encendido en cualquier punto de ejecución[2]. A partir de esta propiedad podemos considerar las secuencias de sentencias P.$s_i; c_1; \cdots; c_n;$ V.s_j dentro de las implementaciones SBD como atómicas. Con el fin de puntualizar la explicación, a esta clase de secuencias las denominaremos *sección* S^{ij}, donde i será el índice del semáforo habilitado por la operación P inicial y j el correspondiente al semáforo modificado por la operación V final.

Regla del dominó: Por la semántica de los semáforos podemos asumir que toda operación V es seguida por una operación P sobre el mismo semáforo, i.e. después que una sección S^{ij} termina, solo puede continuar alguna sección S^{kl} con $j = k$.

Localidad de las variables: Las variables utilizadas por los programas SBD no pueden ser modificadas por otros procesos, i.e. el sistema es cerrado.

La propiedad de exclusión mutua implica que las secciones S^{ij} son ejecutadas una a la vez para todos los procesos. Los posibles "interleavings" entre diferentes procesos son realizados fuera de estas secciones, las cuales pueden considerarse atómicas. Considerando las demás propiedades, el comportamiento del sistema puede ser modelado como una ejecución de las secciones $S^{i_1 j_1}; S^{i_2 j_2}; \cdots; S^{i_k j_k}; \cdots$ con $j_k = i_{k+1}$ lo cual acentúa su carácter secuencial. Otra manera de ver este comportamiento, es pensar el sistema como procesos secuenciales con saltos (sentencias goto) no deterministas entre operaciones V y P sobre el mismos semáforo (saltando entre distintas secciones). Esta última analogía nos servirá para representar el comportamiento con sistemas de transiciones. Cabe aclarar que esta visión secuencial en la ejecución de las secciones no altera la concurrencia del sistema en su conjunto, fuera de las secciones. Los

[2] Una discusión más extensa sobre este punto se encuentra en [Dij79].

programas que ejecutan las regiones críticas condicionales seguirán funcionando de manera concurrente: la ejecución de las secciones es atómica pero fuera de ellas los procesos se comportan concurrentemente. Nuestra modelización hace abstracción de esta concurrencia enfocándose solo en la dinámica de las secciones.

A partir de estas características podemos modelar el comportamiento con un sistema de transiciones de granularidad mayor a los propuestos en [MP91], lo cual simplifica su tratamiento: representaremos cada sección S^{ij} como una transición, identificando con locaciones a los semáforos que se encuentra activos antes y después de la ejecución de la sección (i.e. s_i y s_j para la sección S^{ij}). Más específicamente, cada transición estará asociada con la secuencia de acciones dentro de la región atómica denotada por una sección, y sus locaciones de salida y entrada harán referencia a los semáforos de la sección. Con el fin de simplificar el modelado pediremos que los programas S_i sean sentencias totales y deterministas. La primera restricción es normalmente impuesta sobre los programas de las regiones críticas condicionales para garantizar progreso. La segunda no limita las posibilidades de modelado ya que si un programa S_i posee no determinismo acotado (son programas ejecutables, ver sección 1.5.2, pág. 21), el mismo puede descomponerse en sus partes deterministas y se puede derivar un sistema equivalente construyendo una región crítica condicional para cada una.

Para mostrar como se modelarán las implementaciones SCC_i anotaremos momentáneamente y solo con fines explicativos el programa 4.11 (pág. 121): el comienzo en el texto del programa donde se ejecuta una sección S^{jk} (comenzando en una sentencia P.s_j) se señalará con el texto in_j, y su finalización con el texto out_k (finalizando en una sentencia V.s_k). Para evitar ambigüedades, se primará la anotación correspondiente a la primera operación V.s_m (dentro del **if** inicial) diferenciándola de la que aparece al final (en la ultima guarda del **if** final). Por ejemplo, una sección $S^{mm'}$ comienza su ejecución en in_m y termina en out'_m incrementando el contador y siempre que se satisfaga la guarda $\neg\mathsf{B}_i$. De esta forma, la sección será:

$$\mathrm{P}.s_m; [\neg\mathsf{B}_i]; b_i := b_{i+1}; \mathrm{V}.s_m \quad .$$

Por otro lado, una sección S^{mj} (con $j < m$) comenzará en el mismo punto de ejecución y finalizará en out_j ejecutando el programa S_i, siempre que se verifique la guarda del **if** inicial B_i y la guarda del **if** final $\mathsf{B}_j \wedge b_j > 0$. La sección puede escribirse como:

$$\mathrm{P}.s_m; [\mathsf{B}_i]; \mathsf{S}_i; [\mathsf{B}_j \wedge b_j > 0]; \mathrm{V}.s_j \quad .$$

La idea general para obtener el sistema de transiciones que modele el comportamiento de las implementaciones SBD, será asociar cada sección con una transición. Además, para simplificar la exposición, cada transición será escrita como una sentencia guardad de la forma $\Box\ \mathsf{B} \mapsto s_1; \cdots; s_k$ con $s_1, \cdots, s_k$ sentencias deterministas y totales (i.e. asignaciones). La condición B será calculada propagando el transformador wlp (sección1.4.2, pág. 12) sobre las sentencias que compongan la sección. A continuación enumeraremos todas las transiciones para cada implementación SCC_i diferenciándolas en distintas clases según los puntos de inicio y finalización de cada sección.

Programa 4.11 Componente anotada SCC_i generada por SBD

$$
\begin{array}{ll}
in_m: & \text{P}.s_m; \\
 & \underline{\textbf{if}}\ \ \mathsf{B}_i \mapsto \textbf{skip} \\
 & \,[]\, \neg\mathsf{B}_i \mapsto b_i := b_i + 1\ ; \\
out'_m: & \qquad\qquad \text{V}.s_m; \\
in_i: & \qquad\qquad \text{P}.s_i; \\
 & \qquad\qquad b_i := b_i - 1 \\
 & \underline{\textbf{fi}}; \\
 & \mathsf{S}_i; \\
 & \underline{\textbf{if}}\ \mathsf{B}_0 \wedge b_0 > 0 \mapsto \\
out_0: & \quad \text{V}.s_0 \\
 & \,[]\ \mathsf{B}_1 \wedge b_1 > 0 \mapsto \\
out_1: & \quad \text{V}.s_1 \\
\vdots & \quad \vdots \\
 & \,[]\ \mathsf{B}_j \wedge b_j > 0 \mapsto \\
out_j: & \quad \text{V}.s_j \\
\vdots & \quad \vdots \\
 & \,[]\ \mathsf{B}_{m-1} \wedge b_{m-1} > 0 \mapsto \\
out_{m-1}: & \quad \text{V}.s_{m-1} \\
 & \,[]\ \langle \forall j\ : 0 \leqslant j < m :\ \neg\mathsf{B}_j\ \vee\ b_j = 0 \rangle \mapsto \\
out_m: & \quad \text{V}.m \\
 & \underline{\textbf{fi}}
\end{array}
$$

Clase 1: desde in_m a out'_m.

Las secciones entre estos puntos de inicio y finalización son de la forma

$$\text{P}.s_m; [\neg\mathsf{B}_i]; b_i := b_{i+1}; \text{V}.s_m \quad .$$

Por lo tanto, sus transiciones derivadas serán ejecutada solo si se cumple la condición $\neg\mathsf{B}_i$ lo cual nos dará la guarda de la sentencia en la transición. El estado cambia por el incremento del contador de procesos en espera b_i.

Las m transiciones (una para cada SCC_i) así obtenidas serán:

$$(\ s_m,\ []\ \neg\mathsf{B}_i \mapsto b_i := b_i + 1,\ s_m\ )$$

con $0 \leqslant i < m$.

Clase 2: desde in_m a out_j $(0 \leqslant j < m)$.

Las secciones entre estos puntos de inicio y finalización serán ejecutadas si se cumple B_i en el $\underline{\textbf{if}}$ inicial y $\mathsf{B}_j \wedge b_j > 0$ después del cambio de estado producido por S_i. Por lo tanto pueden escribirse como

$$\text{P}.s_m; [\mathsf{B}_i]; \mathsf{S}_i; [\mathsf{B}_j \wedge b_j > 0]; \text{V}.s_j \quad .$$

Si queremos derivar la transición como una sentencia guardada podemos propagar hacia atrás la guarda final $\mathsf{B}_j \wedge b_j > 0$. Esto puede realizarse aplicando

el transformador wlp en el programa S_i sobre la guarda. Esta transformación es posible gracias a que S_i es determinista. Al finalizar esta clasificación profundizaremos sobre este asunto.

Las transiciones así obtenidas serán:

$$(\, s_m,\ \square\ \mathsf{B}_i \wedge \text{wlp}.\mathsf{S}_i.(\mathsf{B}_j \wedge b_j > 0) \mapsto \mathsf{S}_i,\ s_j \,)$$

con $0 \leqslant i < m$ y $0 \leqslant j < m$.

Clase 3: desde in_m a out_m.

Las secciones en cuestión comenzarán en el mismo punto que las anteriores pero en el caso que ninguna de las primeras m guardas del **<u>if</u>** final se satisfagan:

$$\text{P}.s_m; [\mathsf{B}_i]; \mathsf{S}_i; [\langle \forall j\ : 0 \leqslant j < m :\ \neg \mathsf{B}_j\ \vee\ b_j = 0 \rangle]; \text{V}.s_m\ .$$

De esta forma, para obtener las transiciones asociadas, deberemos propagar la última guarda mediante el transformador wlp sobre el programa S_i:

$$(\, s_m,\ \square\ \mathsf{B}_i \wedge \text{wlp}.\mathsf{S}_i.\langle \forall j\ : 0 \leqslant j < m :\ \neg \mathsf{B}_j\ \vee\ b_j = 0 \rangle \mapsto \mathsf{S}_i,\ s_m \,)$$

con $0 \leqslant i < m$.

Clase 4: desde in_i a out_j $(0 \leqslant i, j < m)$.

Las secciones comenzarán en la sentencia $\text{P}.s_i$ dentro del primer **<u>if</u>** y finalizarán en alguna de las primeras m guardas del **<u>if</u>** final después de alterar el estado por el decremento del contador de procesos b_i y la ejecución de S_i:

$$\text{P}.s_i; b_i := b_i - 1; \mathsf{S}_i; [\mathsf{B}_j \wedge b_j > 0]; \text{V}.s_j\ .$$

Para calcular las guardas de las transiciones asociadas, deberemos aplicar el transformador wlp a las guardas $\mathsf{B}_j \wedge b_j > 0$ sobre el decremento del contador b_i compuesto secuencialmente con el programa S_i. Realizando este cálculo sobre los m programas SCC_i con sus m guardas finales, obtendremos las siguientes transiciones:

$$(\, s_i,\ \square\ \text{wlp}.(b_i := b_i - 1; \mathsf{S}_i).(\mathsf{B}_j \wedge b_j > 0) \mapsto b_i := b_i - 1; \mathsf{S}_i,\ s_j \,)$$

con $0 \leqslant i < m$ y $0 \leqslant j < m$.

Clase 5: desde in_i a out_m $(0 \leqslant i < m)$.

Las secciones comenzarán en el mismo punto que las anteriores pero finalizarán en la última guarda del **<u>if</u>** final:

$$\text{P}.s_i; b_i := b_i - 1; \mathsf{S}_i; [\langle \forall j\ : 0 \leqslant j < m :\ \neg \mathsf{B}_j\ \vee\ b_j = 0 \rangle]; \text{V}.s_m\ .$$

De la misma forma, para obtener las guardas de las transiciones aplicaremos el transformador wlp a la guarda final $\langle \forall j\ : 0 \leqslant j < m :\ \neg \mathsf{B}_j\ \vee\ b_j = 0 \rangle$ sobre el decremento del contador b_i compuesto secuencialmente con el programa S_i:

$$\begin{array}{l}(\, s_i, \\ \square\,\text{wlp}.(b_i := b_i - 1; \mathsf{S}_i).\langle \forall j\ : 0 \leqslant j < m :\ \neg \mathsf{B}_j\ \vee\ b_j = 0 \rangle \mapsto b_i := b_i - 1; \mathsf{S}_i \\ , s_m \,)\end{array}$$

con $0 \leqslant i < m$.

Nota: En el anterior desarrollo se obtuvieron las guardas de las transiciones calculando el transformador wlp de las condiciones finales sobre los programas S_i. Una de las restricciones impuestas es estos programas es que sean deterministas. Veremos que si esta propiedad no se cumple el sistema de transiciones resultante puede no modelar correctamente el comportamiento de las implementaciones SBD. Por ejemplo, sea un programa S_i:

$$\mathsf{S}_i \doteq \underline{\mathbf{if}}\ \mathsf{T} \mapsto x := 0 \ [\!]\ \mathsf{T} \mapsto x := 1\ \underline{\mathbf{fi}}$$

que conforma una sección

$$\mathrm{P}.s_i; [\mathsf{B}_i]; \mathsf{S}_i; [\mathsf{B}_j \wedge b_j > 0]; \mathrm{V}.s_j \quad ,$$

con la condición B_j igual a $x = 0$.

A partir del modelado descripto, la sentencia de la transición asociada a esta sección será:

$$[\!]\ \mathsf{B}_i \wedge \mathrm{wlp}.\mathsf{S}_i.(\mathsf{B}_j \wedge b_j > 0) \mapsto \mathsf{S}_i \quad .$$

Si calculamos el término $\mathrm{wlp}.\mathsf{S}_i.(\mathsf{B}_j \wedge b_j > 0)$ de la guarda anterior obtenemos:

$\quad \mathrm{wlp}.\mathsf{S}_i.(\mathsf{B}_j \wedge b_j > 0)$

$\equiv$ { Definición de B_j }

$\quad \mathrm{wlp}.\mathsf{S}_i.(x = 0 \wedge b_j > 0)$

$\equiv$ { Conjuntividad de wlp }

$\quad \mathrm{wlp}.\mathsf{S}_i.(x = 0) \wedge \mathrm{wlp}.\mathsf{S}_i.(b_j > 0)$

$\equiv$ { Cálculo de wlp con S_i definido }

$\quad \mathsf{T} \Rightarrow 0 = 0 \wedge \mathsf{T} \Rightarrow 1 = 0 \wedge \mathrm{wlp}.\mathsf{S}_i.(b_j > 0)$

$\equiv$ { Lógica de predicados }

$\quad \mathsf{T} \wedge \mathsf{F} \wedge \mathrm{wlp}.\mathsf{S}_i.(b_j > 0)$

$\equiv$ { Lógica de predicados }

$\quad \mathsf{F}$

Por lo tanto la guarda de la transición resultante sera falsa para cualquier estado, y no produce ninguna ejecución posible.

Por otro lado la sección original posee ejecuciones posibles gracias al no determinismo de S_i: dado un scheduler que elija la primer guarda de S_i, B_i será satisfecha. Esta falla en el modelado se debe a que el transformador wlp calcula los estados desde los cuales la ejecución de S_i resulta en otro que cumple la poscondición, cualquiera sea la elección no determinista (o sea para cualquier scheduler).

En el caso que el programa sea determinista, el transformador wlp distribuye con respecto a la implicación, por lo cual la sección y la transición generada son equivalentes:

$\quad \mathrm{wlp}.([\mathsf{B}_i]; \mathsf{S}_i; [\mathsf{B}_j \wedge b_j > 0]).Q$

$\equiv$ { Definición de wlp }

$\quad \mathsf{B}_i \Rightarrow \mathrm{wlp}.\mathsf{S}_i.(\mathsf{B}_j \wedge b_j > 0 \Rightarrow Q)$

$\equiv$ { S_i determinista, wlp distribuye con la implicación (propiedad 1.12.6, pág. 15) }

$\mathsf{B}_i \Rightarrow (\text{wlp}.\mathsf{S}_i.(\mathsf{B}_j \wedge b_j > 0) \Rightarrow \text{wlp}.\mathsf{S}_i.Q)$

$\equiv$ { Lógica de predicados }

$\mathsf{B}_i \wedge \text{wlp}.\mathsf{S}_i.(\mathsf{B}_j \wedge b_j > 0) \Rightarrow \text{wlp}.\mathsf{S}_i.Q$

$\equiv$ { Definición de wlp }

$\text{wlp}.([]\ \mathsf{B}_i \wedge \text{wlp}.\mathsf{S}_i.(\mathsf{B}_j \wedge b_j > 0) \mapsto \mathsf{S}_i).Q$

Nota: Nuestro modelo de regiones críticas con sistema de transiciones no tiene en cuenta los cambios de estado producidos fuera de las mismas ni el orden en que las regiones son ejecutadas: solo se representa el sistema de regiones sin incluir el contexto donde se ejecutan.

Con esta simplificación el sistema de transiciones podría modelar más ejecuciones (sección 1.7, pág. 28) de las que se realizan en el sistema de regiones concreto. Un ejemplo claro de este fenómeno se da en el problema de lectores/escritores (especificado en la sección 4.8.4 más adelante). En el mismo, un lector (o escritor) ejecuta la región de salida solo si antes ejecutó la de entrada. Como el modelado se hace sobre cada región por separado y sin tener en cuenta la relación entre las mismas, habrá ejecuciones de la región de salida sin haber antes una sobre la entrada. De esta manera, se incluirán ejecuciones en el sistema de transiciones que no pertenecen al problema concreto.

Este tipo de abstracción realizada en el modelado no altera la corrección del método propuesto, ya que al incluir nuevas ejecuciones los invariantes del sistema de transiciones serán más débiles que los del problema concreto. Esta posible relajación de los invariantes no impidió encontrar la solución en los problemas donde se aplicó nuestro método (sección 4.8).

Cabe destacar, que la simplificación lograda con esta abstracción es central para poder resolver el problema ya que modelar todo el contexto resultaría en sistemas de transiciones de tamaño demasiado grande para ser tratados. $\square$

Las transiciones enumeradas forman el conjunto $\mathcal{T}$ del sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ que modela el comportamiento de una implementación SBD. Como ya se mencionó, el conjunto de locaciones $\mathcal{L}$ estará identificado con el conjunto de semáforos:

$$\mathcal{L} \ \doteq\ \{s_i \mid 0 \leqslant i \leqslant m\}$$

y las configuraciones iniciales Θ serán:

$$\begin{aligned} \Theta.s_i &\ \doteq\ \mathsf{false} \quad \text{si} \quad 0 \leqslant i < m\ , \\ \Theta.s_m &\ \doteq\ \theta \wedge \langle\, \forall\, j\ : 0 \leqslant j < m :\ b_i = 0\, \rangle \end{aligned}$$

con θ la condición inicial del problema. El conjunto de sentencias $\mathcal{S}$ estará formado por los comandos guardados de las transiciones.

A partir del invariante brindado por la técnica SBD en la ecuación 4.2 (pág. 110), podemos obtener un invariante inicial para el sistema de transiciones generado. Notar que cada término conjuntivo de aquel, correspondiente a un semáforo, es el predicado que se cumple si el semáforo se encuentra activo, es decir, al momento de ejecutar las transiciones que salen del semáforo. De esta

forma, cada uno de estos términos será el invariante en la locación identificada con el semáforo:

$$\begin{aligned} \varphi_{SBD}.s_i &\doteq \mathsf{B}_i \wedge b_i > 0 \wedge I \quad \text{si} \quad 0 \leqslant i < m \\ \varphi_{SBD}.s_m &\doteq \langle \forall j : 0 \leqslant j < m : \neg \mathsf{B}_j \vee b_j = 0 \rangle \wedge \\ &\qquad \langle \forall j : 0 \leqslant j < m : b_j \geqslant 0 \rangle \wedge I \end{aligned} \tag{4.3}$$

con I el invariante del problema.

Ejemplo 4.2 (Productor/Consumidor con sistema de transiciones) A manera de ejemplo, veremos como representar con sistemas de transiciones la solución SBD del problema Productor/Consumidor sobre buffer acotado (ejemplo anterior en pág.111). La implementación del componente productor se muestra en el programa 4.12. Para adecuarnos al esquema general anteriormente

Programa 4.12 Productor anotado

```
in₂:   P.s₂;
       if  n < N ↦ skip
       ▯ ¬n < N ↦ b₀ := b₀ + 1;
out'₂:              V.s₂;
in₀:                P.s₀;
                    b₀ := b₀ − 1
       fi;
       n := n + 1;
       if n < N ∧ b₀ > 0 ↦
out₀:     V.s₀
       ▯ n > 0 ∧ b₁ > 0  ↦
out₁:     V.s₁
       ▯ (n = N ∨ b₀ = 0) ∧ (n = 0 ∨ b₁ = 0) ↦
out₂:     V.s₂
       fi
```

explicado, hemos reemplazado los semáforos s (semáforo de bloqueo del productor), t (semáforo de bloqueo del consumidor) y m (el semáforo neutral) por s_0, s_1 y s_2 respectivamente. También reemplazamos los contadores b (contador de esperas de productores) y c (contador de esperas de consumidores) por b_0 y b_1 respectivamente. Las guardas también fueron reemplazadas para seguir el esquema general (en el programa 4.11) de las implementaciones SBD.

Comenzando en in_2 (operación P.s_2 al principio del programa) tenemos cuatro posibles secciones que terminan en out'_2, out_0, out_1 y out_2. La primera se traducirá a una transición de clase 1, las dos siguientes serán de clase 2 y la última será de clase 3. Las guardas de estas transiciones serán obtenidas buscando la condición que se debe satisfacer para ejecutar la sección correspondiente. Tomemos como ejemplo la sección que finaliza en out'_2:

$$\text{P}.s_2; [\neg n < N]; b_0 := b_0 + 1; \text{V}.s_2 \ .$$

Al principio de su ejecución se deberá satisfacer la guarda $\neg n < N$. Por lo tanto la condición de la asignación guardad de la transición sera exactamente esta guarda. La parte de la asignación, en esta sentencia de la transición,

será obtenida directamente por el cambio de estado producido por la sección correspondiente. En este caso, solo tenemos la asignación $b := b + 1$ como posible sentencia de cambio de estado. Además, las locaciones de salida y entrada estarán representadas por el semáforo inicial en la operación P y el final en la operación V respectivamente. De esta forma la transición en cuestión será:

$$(s_2,\ \square\ \neg n < N \mapsto b_0 := b_0 + 1,\ s_2)\ \ .$$

Continuaremos con las tres secciones siguientes que comienzan en in_2:

$$\begin{aligned}
&\mathrm{P}.s_2; [n < N]; n := n + 1; [n < N \wedge b_0 > 0]; \mathrm{V}.s_0\ \ ,\\
&\mathrm{P}.s_2; [n < N]; n := n + 1; [n > 0 \wedge b_1 > 0]; \mathrm{V}.s_1\ \ ,\\
&\mathrm{P}.s_2; [n < N]; n := n + 1; [(n = N\ \vee\ b_0 = 0) \wedge (n = 0\ \vee\ b_1 = 0)]; \mathrm{V}.s_2\ \ .
\end{aligned}$$

Para ejecutar cualquiera de ellas debe cumplirse la guarda $n < N$ del **<u>if</u>** inicial. Además, cada sección finaliza verificando las condiciones en el **<u>if</u>** final. De esta forma, las transiciones podrán ejecutarse solo si se satisface alguna de estas condiciones al finalizar su ejecución. Como la definición de la asignación guardada en la transición requiere que su guarda sea verificada antes de la asignación, propagaremos con el transformador wlp sobre la sección. Al igual que con la transición obtenida anteriormente, las locaciones de entrada y salida serán los semáforos de las operaciones P y V. Por ejemplo, de la sección que termina en out_0 obtendremos la siguiente transición de clase 2:

$$(s_2,\ \square\ n < N \wedge \mathrm{wlp}.(n := n + 1).(n < N \wedge b_0 > 0) \mapsto n := n + 1,\ s_0)\ \ .$$

Calculando el resultado del transformador wlp y simplificando la guarda, podemos escribir la transición como:

$$(s_2,\ \square\ n + 1 < N \wedge b_0 > 0 \mapsto n := n + 1,\ s_0)\ \ .$$

El resto de las transiciones que comienzan en in_2 es calculado de la misma manera.

Existen además tres secciones que comienzan en la operación P dentro del **<u>if</u>** inicial (en in_0) y terminan en alguna operación V dentro del **<u>if</u>** final (out_0, out_1 u out_2):

$$\begin{aligned}
&\mathrm{P}.s_0; b_0 := b_0 - 1; n := n + 1; [n < N \wedge b_0 > 0]; \mathrm{V}.s_0\ \ ,\\
&\mathrm{P}.s_0; b_0 := b_0 - 1; n := n + 1; [n > 0 \wedge b_1 > 0]; \mathrm{V}.s_1\ \ ,\\
&\mathrm{P}.s_0; b_0 := b_0 - 1; n := n + 1; [(n = N\ \vee\ b_0 = 0) \wedge (n = 0\ \vee\ b_1 = 0)]; \mathrm{V}.s_2\ \ .
\end{aligned}$$

Las guardas de las asignaciones en las transiciones generadas por esta secciones serán obtenidas propagando, con el transformador wlp, las condiciones del **<u>if</u>** final sobre las sentencias de las secciones, de la misma forma que se explicó anteriormente. Estas secciones ejecutan el decremento del contador de procesos y el incremento de la variable n. Las asignaciones de las secciones serán modeladas como un asignación múltiple. Por ejemplo, de la sección que termina en out_1 (segunda sección anterior, donde se despierta un consumidor) se derivará la siguiente transición de clase 4:

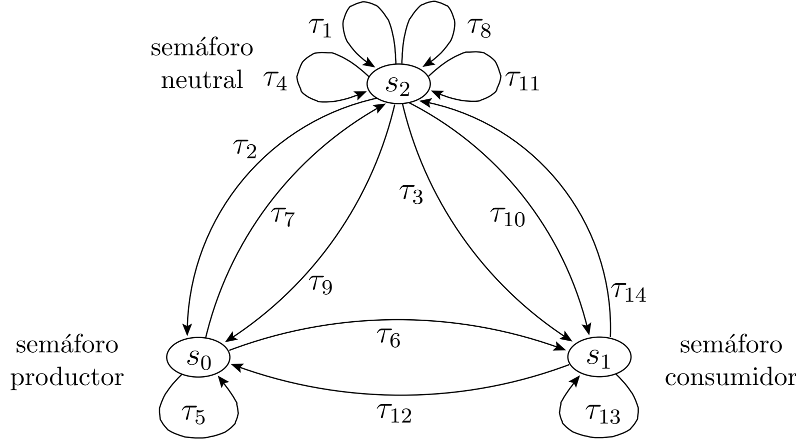
$$(s_0,\ \square\ \mathrm{wlp}.(b_o, n := b_0 - 1, n + 1).(n > 0 \wedge b_1 > 0) \mapsto b_0, n := b_0 - 1, n + 1,\ s_1)$$

y calculando el resultado del transformador wlp obtendremos:

$$(s_0,\ \square\ n+1>0 \wedge b_1>0 \mapsto b_0, n := b_0-1, n+1,\ s_1)\ .$$

Las transiciones correspondiente al consumidor se obtienen de manera análoga. En el gráfico 4.1 se muestran todas las transiciones así calculadas.

Grafo de transiciones 4.1 Sistema de transiciones Productor/Consumidor



Transiciones Productor:

$\tau_1:\quad (s_2,\ \square\ \neg n<N \mapsto b_0 := b_0+1,\ s_2)\ ,$

$\tau_2:\quad (s_2,\ \square\ n+1<N \wedge b_0>0 \mapsto n := n+1,\ s_0)\ ,$

$\tau_3:\quad (s_2,\ n<N \wedge n+1>0 \wedge b_1>0 \mapsto n := n+1,\ s_1)\ ,$

$\tau_4:\quad (s_2,\ n<N \wedge (\neg n+1<N \ \vee\ b_0=0) \wedge (\neg n+1>0 \ \vee\ b_1=0)$
$\mapsto n := n+1,\ s_2)\ ,$

$\tau_5:\quad (s_0,\ n+1<N \wedge b_0-1>0 \mapsto b_0, n := b_0-1, n+1,\ s_0)\ ,$

$\tau_6:\quad (s_0,\ n+1>0 \wedge b_1>0 \mapsto b_0, n := b_0-1, n+1,\ s_1)\ ,$

$\tau_7:\quad (s_0,\ (\neg n+1<N \ \vee\ b_0-1=0) \wedge (\neg n+1>0 \ \vee\ b_1=0)$
$\mapsto b_0, n := b_0-1, n+1,\ s_2)$

Transiciones Consumidor:

$\tau_8:\quad (s_2,\ \neg n>0 \mapsto b_1 := b_1+1,\ s_2)\ ,$

$\tau_9:\quad (s_2,\ n>0 \wedge n-1<N \wedge b_0>0 \mapsto n := n-1,\ s_0)\ ,$

$\tau_{10}:\quad (s_2,\ n-1>0 \wedge b_1>0 \mapsto n := n-1,\ s_1)\ ,$

$\tau_{11}:\quad (s_2,\ n>0 \wedge (\neg n-1<N \ \vee\ b_0=0) \wedge (\neg n-1>0 \ \vee\ b_1=0)$
$\mapsto n := n-1,\ s_2)\ ,$

$\tau_{12}:\quad (s_1,\ n-1<N \wedge b_0>0 \mapsto b_1, n := b_1-1, n-1,\ s_0)\ ,$

$\tau_{13}:\quad (s_1,\ n-1>0 \wedge b_1-1>0 \mapsto b_1, n := b_1-1, n-1,\ s_1)\ ,$

$\tau_{14}:\quad (s_1,\ (\neg n-1<N \ \vee\ b_0=0) \wedge (\neg n-1>0 \ \vee\ b_1-1=0)$
$\mapsto b_1, n := b_1-1, n-1,\ s_2)$

Estas transiciones conforman el conjunto $\mathcal{T}$ del sistema $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ que modela el comportamiento de los procesos consumidores y productores. Además, el conjunto de sentencias $\mathcal{S}$ serán las asignaciones guardadas en las

transiciones, el conjunto de locaciones serán los tres semáforos $\mathcal{L} \doteq \{s_0, s_1, s_3\}$ y el conjunto de configuraciones iniciales será:

$$\Theta \doteq [\mathsf{false}, \mathsf{false}, n = 0 \wedge N > 0 \wedge b_0 = 0 \wedge b_1 = 0]$$

($n = 0 \wedge N > 0$ es la condición inicial del problema θ).

Como ya se mencionó, la técnica SBD provee además el siguiente invariante del sistema de transiciones:

$$\begin{aligned} \varphi_{SBD} \doteq [\ & 0 \leqslant n < N \wedge b_0 > 0,\ 0 < n \leqslant N \wedge b_1 > 0, \\ & (\neg n < N \ \vee\ b_0 = 0) \wedge (\neg n > 0 \ \vee\ b_1 = 0) \\ & \wedge\ 0 \leqslant n \leqslant N \wedge b_0 \geqslant 0 \wedge b_1 \geqslant 0\ ]\ . \end{aligned}$$

4.5 Eliminación de guardas

En esta sección comenzaremos a desarrollar el método para eliminar guardas finales en las implementaciones SBD. Como veremos, el mismo no solo encuentra estas simplificaciones, si no que además detecta la imposibilidad de eliminación de aquellas guardas. La técnica propuesta es incompleta, en el sentido que puede no detectar guardas cuya eliminación es correcta, sin embargo en la mayoría de los ejemplos de la literatura se detectaron todas las simplificaciones posibles. Cabe remarcar que esta incompletitud no es un problema a la hora de obtener programas correctos: las implementaciones de regiones críticas condicionales producidas por la técnica SBD sin estas simplificaciones son correctas aunque menos eficientes. Como se mencionó al inicio de este capítulo, el objetivo de aplicación del método es en el desarrollo de compiladores y generadores de código para aumentar la eficiencia de construcciones concurrentes de alto nivel, por lo cual esta deficiencia no es un problema: los programas generados son correctos aunque no se detecten todas las mejoras posibles. De todas maneras, al final de esta sección se desarrollará un análisis de las causas de este fenómeno y se mostrará como fueron tratadas.

Dado el sistema de transiciones que modela la implementación SBD de un problema de regiones críticas condicionales, aplicaremos principalmente la técnica de propagación hacia atrás desarrollada en la sección 2.3.2 (pág. 53), con el fin de probar de manera automática la factibilidad de eliminación de guardas finales superfluas. Para explorar esta posibilidad utilizaremos el teorema 2.24 (pág. 54) mediante el cual, la invariancia de un predicado P es garantizada si $\Theta \subseteq \nu.\mathcal{B}_{\mathcal{T},P}$ (las configuraciones iniciales implican el máximo punto fijo del transformador $\mathcal{B}_{\mathcal{T},P}$). Tomando el predicado P como el invariante candidato que denote la imposibilidad de ejecución de una guarda, si aquella implicación es válida entonces la guarda puede ser eliminada.

El invariante candidato P será calculado mediante el transformador wlp sobre las transiciones entrantes a la locación donde se ejecuta la guarda. Estos transformadores, aplicados al predicado false, devolverán las configuraciones P desde las cuales la posible guarda a eliminar no se ejecuta. Por lo tanto, si P es invariante la guarda puede ser eliminada.

Tomemos la guarda $B_j \wedge b_j > 0$ en el programa 4.11 (asociada a la operación V.s_j en la anotación out_j). Esta guarda pertenece a las dos transiciones obtenidas del programa SCC_i cuya locación de llegada es s_j, la primera de clase 2 y la

segunda de clase 4:

$$(s_m,\ \square\ \mathsf{B}_i \wedge \text{wlp}.\mathsf{S}_i.(\mathsf{B}_j \wedge b_j > 0) \mapsto \mathsf{S}_i,\ s_j\ )$$
$$(s_i,\ \square\ \text{wlp}.(b_i := b_i - 1; \mathsf{S}_i).(\mathsf{B}_j \wedge b_j > 0) \mapsto b_i := b_i - 1; \mathsf{S}_i,\ s_j\ )$$

Aplicando el transformador wlp, sobre la sentencia de la primer transición, al predicado false, tenemos:

$$
\begin{array}{cl}
 & \text{wlp}.(\square\ \mathsf{B}_i \wedge \text{wlp}.\mathsf{S}_i.(\mathsf{B}_j \wedge b_j > 0) \mapsto \mathsf{S}_i).\mathsf{false} \\
\equiv & \{ \text{ Definición de wlp } \} \\
 & \mathsf{B}_i \wedge \text{wlp}.\mathsf{S}_i.(\mathsf{B}_j \wedge b_j > 0) \Rightarrow \text{wlp}.\mathsf{S}_i.\mathsf{false} \\
\equiv & \{ \ \mathsf{S}_i \text{ es total entonces wlp}.\mathsf{S}_i \text{ es estricto (propiedad 1.12.5, pág. 15) } \} \\
 & \mathsf{B}_i \wedge \text{wlp}.\mathsf{S}_i.(\mathsf{B}_j \wedge b_j > 0) \Rightarrow \mathsf{false} \\
\equiv & \{ \text{ Lógica de predicados } \} \\
 & \mathsf{B}_i \Rightarrow \neg\, \text{wlp}.\mathsf{S}_i.(\mathsf{B}_j \wedge b_j > 0) \\
\equiv & \{ \ \mathsf{S}_i \text{ es total y determinista entonces wlp}.\mathsf{S}_i \text{ distribuye con la negación} \\
 & \quad \text{(propiedad 1.12.7) } \} \\
 & \mathsf{B}_i \Rightarrow \text{wlp}.\mathsf{S}_i.\neg(\mathsf{B}_j \wedge b_j > 0)
\end{array}
$$

Notar que este resultado es el conjunto de estados tal que, mediante la transformación determinista de la transición, se llega a la negación de la guarda a eliminar. De esta manera, el predicado P' que denota las configuraciones desde las cuales la primer transición no se ejecuta será este resultado en la locación s_m (locación de salida de la transición en cuestión) y true en las demás componentes (ya que desde esas locaciones la transición no se ejecuta):

$$
\begin{array}{l}
P'.s_m \doteq \mathsf{B}_i \Rightarrow \text{wlp}.\mathsf{S}_i.\neg(\mathsf{B}_j \wedge b_j > 0) \\
P'.s_k \doteq \mathsf{true} \quad \text{si} \quad 0 \leqslant k < m \ .
\end{array}
$$

Aplicando el mismo razonamiento sobre la segunda transición tenemos:

$$
\begin{array}{cl}
 & \text{wlp}.(\square\ \text{wlp}.(b_i := b_i - 1; \mathsf{S}_i).(\mathsf{B}_j \wedge b_j > 0) \mapsto b_i := b_i - 1; \mathsf{S}_i).\mathsf{false} \\
\equiv & \{ \text{ Definición de wlp } \} \\
 & \text{wlp}.(b_i := b_i - 1; \mathsf{S}_i).(\mathsf{B}_j \wedge b_j > 0) \Rightarrow \text{wlp}.(b_i := b_i - 1; \mathsf{S}_i).\mathsf{false} \\
\equiv & \{ \ b_i := b_i - 1; \mathsf{S}_i \text{ es total entonces wlp}.(b_i := b_i - 1; \mathsf{S}_i) \text{ es estricto (pro-} \\
 & \quad \text{piedad 1.12.5) } \} \\
 & \text{wlp}.(b_i := b_i - 1; \mathsf{S}_i).(\mathsf{B}_j \wedge b_j > 0) \Rightarrow \mathsf{false} \\
\equiv & \{ \text{ Lógica de predicados } \} \\
 & \neg\, \text{wlp}.(b_i := b_i - 1; \mathsf{S}_i).(\mathsf{B}_j \wedge b_j > 0) \\
\equiv & \{ \ b_i := b_i - 1; \mathsf{S}_i \text{ es total y determinista entonces wlp}.(b_i := b_i - 1; \mathsf{S}_i) \\
 & \quad \text{distribuye con la negación (propiedad 1.12.7) } \} \\
 & \text{wlp}.(b_i := b_i - 1; \mathsf{S}_i).\neg(\mathsf{B}_j \wedge b_j > 0)
\end{array}
$$

De igual manera que con la transición anterior, este resultado es la aplicación del transformador wlp a la negación de la guarda a eliminar, sobre las sentencias de la sección anteriores a la ejecución de la guarda.

El predicado P'' que denota las configuraciones desde las cuales la segunda transición no se ejecuta será este resultado en la locación s_i (locación de salida de la transición en cuestión) y true en las demás componentes:

$$\begin{aligned} P''.s_i &\doteq \text{wlp}.(b_i := b_i - 1; \mathsf{S}_i).\neg(\mathsf{B}_j \wedge b_j > 0) \\ P''.s_k &\doteq \mathsf{true} \quad \text{si} \quad 0 \leqslant k \leqslant m \wedge k \neq i \ . \end{aligned}$$

En consecuencia, para poder eliminar una guarda $\mathsf{B}_j \wedge b_j > 0$ (con $0 \leqslant j < m$) en un programa SCC_i (con $0 \leqslant i < m$), a partir predicados P' y P'' se puede construir el invariante candidato P como la conjunción $P' \cap P''$:

$$\begin{aligned} P.s_i &\doteq \text{wlp}.(b_i := b_i - 1; \mathsf{S}_i).\neg(\mathsf{B}_j \wedge b_j > 0) \\ P.s_m &\doteq \mathsf{B}_i \Rightarrow \text{wlp}.\mathsf{S}_i.\neg(\mathsf{B}_j \wedge b_j > 0) \\ P.s_k &\doteq \mathsf{true} \quad \text{si} \quad 0 \leqslant k < m \wedge k \neq i \ , \end{aligned} \tag{4.4}$$

ya que el conjunto de configuraciones desde los cuales la guarda no se ejecuta (dentro de ambas transiciones) es la intersección entre las configuraciones en P' y P''.

De esta manera obtenemos el invariante candidato P que denota la imposibilidad de ejecución de una de las m primeras guardas en uno de los programas S_i. Las guardas finales, que tienen la forma $\langle \forall j : 0 \leqslant j < m : \neg \mathsf{B}_j \vee b_j = 0 \rangle$ no pueden ser eliminadas, ya que son las que posibilitan la liberación del semáforo neutral s_m que permite el progreso del sistema: si un proceso no puede satisfacer las guardas anteriores, debe permitir el ingreso de nuevos procesos al sistema.

Con el invariante candidato P y el sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ definidos podemos aplicar de demostración de k-invariancia esquematizado en el programa 2.3 (pág. 62) para verificar si P es un invariante. Este método (con las modificaciones que explicaremos luego) será aplicado a cada guarda de cada programa S_i ($m \times m$ guardas). Como ya se mencionó en esa sección, el método detecta además la imposibilidad de invariancia de P, o lo que es equivalente, la imposibilidad de eliminar la guarda en cuestión.

Como ya dijimos, el método no es completo (la detección de invariancia no está asegurada). Las causas de esta incompletitud son las siguientes:

1. la cadena descendente de fórmulas en el cálculo del mayor punto fijo intermedias es infinita (como se explica en la sección 2.3.4 pág. 62) y el método de demostración de k-invariancia no termina.

2. El procedimiento de decisión sobre el orden de predicados no es completo en la verificación de las guardas del bucle del programa 2.3.

3. El crecimiento en la representación de los predicados envueltos en el programa anterior hace que sea imposible calcular el máximo punto fijo en tiempos razonables.

La primer causa es inherente al método de cálculo del punto fijo. Debido a este fenómeno se utilizó el método de propagación hacia atrás, que en contraste con el método de propagación hacia adelante, la cadena de predicados intermedios hacia el punto fijo es usualmente finita (sección 2.3.4). Esta característica fue corroborada en la mayor parte de los ejemplos de la literatura donde se

aplicó nuestro método (más adelante veremos estos resultados). Otra importante ventaja es que las fórmulas intermedias son libres de cuantificadores, lo cual beneficia el funcionamiento completo de probadores teoremas automáticos para verificar las implicaciones envueltas en el programa. Cabe agregar que si una guarda no puede ser eliminada, la cadena de predicados intermedios hasta detectar esta situación es finita, como se demuestra en la sección 2.3.4. Para el caso que la cadena sea infinita, se puede agregar al programa 2.3 un contador que limite la cantidad de iteraciones o un límite de tiempo de procesamiento (time out). En los casos que esto suceda el algoritmo termina sin poder decidir la invariancia del predicado. Como ya se mencionó, esto no es un problema grave ya que el programa original es correcto aunque no se detecte la guarda a eliminar.

La segunda causa es producida por la incompletitud de los probadores externos utilizados para decidir las relaciones de orden en la guarda del bucle en el método. Para tratar este fenómeno, se diseñó el prototipo de software que implementa el método añadiendo la posibilidad de utilizar varios probadores externos en paralelo. Esto aumenta las posibilidades de verificación de fórmulas, ya que se aprovecha las características propias de cada uno para resolver el problema. Por esta razón, el prototipo tiene la posibilidad de emplear el SMT solver CVC3, y los probadores de teoremas Isabelle/HOL y ACL2 [ACL08]. En la práctica, los problemas de regiones críticas condicionales derivan en sistemas de transiciones lineales en el sentido que se explica en la sección 3.3.3 (pág. 93) (todos los ejemplos de la literatura probados tienen esta característica), por lo cual es suficiente utilizar solo CVC3 como probador externo ya que es completo para la aritmética lineal.

De todas maneras, para detectar la imposibilidad de verificación de una fórmula, se modificó la semántica de aquella guarda manteniendo corrección: el método termina solo cuando puede ser probada alguna de las negaciones de las guardas $\Theta \subseteq B_k$ y $\neg(B_k \subseteq \mathcal{B}_{\mathcal{T},P}.B_k)$. Formalmente, si indicamos con el símbolo $\vdash$ que se encontró una prueba, el método termina cuando se demuestra la no satisfabilidad de $\Theta \subseteq B_k$ o la validez de $B_k \subseteq \mathcal{B}_{\mathcal{T},P}.B_k$:

$$\vdash \neg(\Theta \subseteq B_k) \;\; \vee \;\; \vdash B_k \subseteq \mathcal{B}_{\mathcal{T},P}.B_k \;\; .$$

De esta forma, la guarda del bucle será la negación de esta condición y el método decidirá la invariancia solo si se puede demostrar $\vdash \Theta \subseteq B_k$, como se indica en el programa 4.13 (pág. 132). Notar con respecto a esto último, que si se puede probar esta implicación, por la precondición la cadena es finita.

La tercer causa de incompletitud es producida por el crecimiento en la representación de los elementos de la cadena calculada. Esto se debe a la aplicación reiterada del transformador de propagación $\mathcal{B}_{\mathcal{T},P}$ que se hace en cada iteración, lo cual produce un crecimiento en la representación del predicados intermedios (almacenados en la variable B_k) a medida que se calcula la cadena. Como resultado de este fenómeno, la verificación de las implicaciones que procesan los probadores externos demora más tiempo a medida que el método progresa. Este comportamiento fue verificado, incluso en ejemplos de la literatura muy simples, donde a las pocas iteraciones se saturan los recursos computacionales por falta de memoria. Por esta razón, el empleo de distintas estrategias es esencial para la aplicabilidad del método y por ende al objetivo del trabajo. El tratamiento de este problema será explayado en la sección siguiente.

Programa 4.13 Propagación hacia atrás con probadores externos

es_invariante (P, Θ : $\mathsf{Pred}_{\mathcal{L}}$; $\mathcal{T}$: $\mathsf{Tran}_{\mathcal{L},\mathcal{S}}$) : Bool
$\mathcal{B}_{\mathcal{T},P} := \langle \lambda X \bullet P \cap \text{WLP}.\mathcal{T}.X \rangle;$
$B_k := \mathsf{True};$
do $\neg(\vdash \neg(\Theta \subseteq B_k)) \wedge \neg(\vdash B_k \subseteq \mathcal{B}_{\mathcal{T},P}.B_k)$
$B_k := \mathcal{B}_{\mathcal{T},P}.B_k$
od;
$\{\vdash \neg(\Theta \subseteq B_k) \;\vee\; \vdash B_k \subseteq \mathcal{B}_{\mathcal{T},P}.B_k\}$
return $\vdash \Theta \subseteq B_k$

4.6 Mejoras del método

Como acabamos de mencionar, el crecimiento de las fórmulas intermedias en el cálculo del punto fijo suele producir un estancamiento en su procesamiento haciendo inviable la aplicabilidad del método. Para contrarrestar este fenómeno se utilizaron diversas estrategias:

1. Se utilizó el invariante φ_{SBD} producido por la técnica SBD.
2. Se simplificaron los sistemas de transiciones.
3. Se simplificó el invariante candidato P.
4. Se emplearon métodos de simplificación de predicado, propios y de probadores externos.

La aplicación conjunta de todas ellas produjo una drástica aceleración del método, obteniéndose el resultado final en segundos, cuando sin aplicarlas el método tardó días e incluso no terminó por agotamiento total de los recursos computacionales. A continuación detallaremos cada una de estas estrategias.

4.6.1 Simplificación con invariante SBD

La técnica SBD genera de forma automática este invariante, el cual es válido cualquiera sean las guardas a eliminar. Utilizándolo como premisas de las implicaciones en la guarda del bucle, se acelera la verificación de las mismas. En este sentido, los probadores de teoremas externos permiten el ingreso de premisas que se asumen válidas al momento de intentar probar la validez de un predicado. De esta manera se reemplazaron las fórmulas en las guardas y en el valor devuelto agregando esta suposición, como se muestra en el programa 4.14 (pág. 133).

Notar que la idea de emplear esta estrategia surge del proceso manual de eliminación de guardas visto en el ejemplo 4.1. En este ejemplo, se comienza con el invariante φ_{SBD} el cual se va fortaleciendo a medida que se propaga la negación de la guarda. En el último programa la detección del punto fijo es realizada por la prueba de $\varphi_{SBD} \vdash B_k \subseteq \mathcal{B}_{\mathcal{T},P}.B_k$, lo cual es equivalente a probar $\vdash (\varphi_{SBD} \cap B_k) \subseteq \mathcal{B}_{\mathcal{T},P}.B_k$. Esta última fórmula muestra como en cada paso de la iteración se fortalece el invariante φ_{SBD} hasta llegar al punto fijo.

Programa 4.14 Propagación hacia atrás utilizando φ_{SBD}

es_invariante (P, Θ : $\mathsf{Pred}_{\mathcal{L}}$; $\mathcal{T}$: $\mathsf{Tran}_{\mathcal{L},\mathcal{S}}$) : Bool

$\mathcal{B}_{\mathcal{T},P} := \langle \lambda X \bullet P \cap \text{WLP}.\mathcal{T}.X \rangle;$

$B_k := \mathsf{True};$

do $\neg(\varphi_{SBD} \vdash \neg(\Theta \subseteq B_k)) \wedge \neg(\varphi_{SBD} \vdash B_k \subseteq \mathcal{B}_{\mathcal{T},P}.B_k)$

$B_k := \mathcal{B}_{\mathcal{T},P}.B_k$

od;

return $\varphi_{SBD} \vdash \Theta \subseteq B_k$

Como veremos, el invariante φ_{SBD} fue también utilizado en las demás estrategias.

4.6.2 Simplificación del sistema de transiciones

En la sección 4.4 vimos como generar un sistema de transiciones a partir de una solución SBD. En esta sección veremos que algunas de las transiciones generadas pueden ser eliminadas, ya que no contribuyen al cálculo del punto fijo, o sus guardas simplificadas cualquiera sea el problema a tratar.

Eliminación de transiciones no ejecutadas

Dado uno de los m programas SCC_i generado por la técnica (programa 4.11, pág. 121), las transiciones de clase 2 que van de in_m (principio del programa) a out_j (**if** final), en el caso $j = i$ (el proceso libera a otro ejecutando el mismo programa) tienen la forma:

$$(\, s_m,\ \square\ \mathsf{B}_i \wedge \text{wlp}.\mathsf{S}_i.(\mathsf{B}_i \wedge b_i > 0) \mapsto \mathsf{S}_i,\ s_i\,)\ ,$$

con $0 \leqslant i < m$. Analicemos las guardas de las sentencias en estas transiciones, teniendo en cuenta que el programa S_i no modifica los contadores b_i (son variables auxiliares agregadas por la técnica SBD):

$$\begin{aligned}
& \mathsf{B}_i \wedge \text{wlp}.\mathsf{S}_i.(\mathsf{B}_i \wedge b_i > 0) \\
\equiv\ & \{\text{ Conjuntividad de wlp }\} \\
& \mathsf{B}_i \wedge \text{wlp}.\mathsf{S}_i.\mathsf{B}_i \wedge \text{wlp}.\mathsf{S}_i.(b_i > 0) \\
\equiv\ & \{\ \mathsf{S}_i \text{ no modifica los contadores }\} \\
& \mathsf{B}_i \wedge \text{wlp}.\mathsf{S}_i.\mathsf{B}_i \wedge b_i > 0 \\
\equiv\ & \{\text{ Conmutatividad y asociatividad de } \wedge \} \\
& (\mathsf{B}_i \wedge b_i > 0) \wedge \text{wlp}.\mathsf{S}_i.\mathsf{B}_i
\end{aligned}$$

Por otro lado, la componente m-ésima (en la locación s_m, salida de la transición anterior) del invariante φ_{SBD} (ecuación 4.3, pag. 125) es:

$$\begin{aligned}
\varphi_{SBD}.s_m \ \doteq\ & \langle \forall j \ : 0 \leqslant j < m :\ \neg\mathsf{B}_j \ \vee\ b_j = 0 \,\rangle \wedge \\
& \langle \forall j \ : 0 \leqslant j < m :\ b_j \geqslant 0 \,\rangle \wedge I\ \ ,
\end{aligned}$$

donde el término conjuntivo $\neg\mathsf{B}_i \lor b_i = 0$ niega la guarda anterior. De esta forma, como φ_{SBD} es invariante, nunca se satisfará la guarda de estas transiciones y por lo tanto no se ejecutarán. En consecuencia, las m transiciones pueden ser eliminadas del sistema.

Eliminación de transiciones utilizando el invariante candidato

Como veremos, también pueden ser eliminadas otras transiciones dado un invariante candidato P fijo. Consideremos el caso en que este invariante candidato represente la posibilidad de eliminación de una guarda $\mathsf{B}_k \land b_k > 0$ en un programa SCC_i, con lo cual P en las locaciones s_m y s_i, por la ecuación 4.4, estará definido como:

$$
\begin{aligned}
P.s_m &\doteq \mathsf{B}_i \Rightarrow \mathrm{wlp}.\mathsf{S}_i.\neg(\mathsf{B}_k \land b_k > 0)\\
P.s_i &\doteq \mathrm{wlp}.(b_i := b_i - 1; \mathsf{S}_i).\neg(\mathsf{B}_k \land b_k > 0) \ .
\end{aligned}
$$

De manera intuitiva, puede verse que las ejecuciones del sistema que hacen verdadero este predicado no pueden ejecutar las transiciones que contienen la guarda $\mathsf{B}_k \land b_k > 0$, por lo tanto estas transiciones pueden ser eliminadas en el cálculo del punto fijo. Formalmente, en el método de demostración de invariancia se calcula el transformador $\mathcal{B}_{\mathcal{T},P}$ definido como la conjunción $P \cap \mathrm{WLP}.\mathcal{T}.X$. Veremos que el invariante candidato $P.s_m$ es más fuerte que el resultado del transformador en la transición que contienen la guarda a eliminar y con locación de salida s_m. La única transición que cumple este requisito es de clase 2 (desde in_m a out_j):

$$(s_m,\ \square\ \mathsf{B}_i \land \mathrm{wlp}.\mathsf{S}_i.(\mathsf{B}_k \land b_k > 0) \mapsto \mathsf{S}_i,\ s_k)$$

El término conjuntivo del resultado de $\mathrm{WLP}.\mathcal{T}.X$ correspondiente a esta transición (ver definición de WLP en la sección 1.6.2, pág. 24) es:

$$\mathsf{B}_i \land \mathrm{wlp}.\mathsf{S}_i.(\mathsf{B}_k \land b_k > 0) \Rightarrow \mathrm{wlp}.\mathsf{S}_i.X \ .$$

Veremos entonces que $P.s_m$ es más fuerte que este resultado:

$$
\begin{aligned}
& P.s_m \Rightarrow (\mathsf{B}_i \land \mathrm{wlp}.\mathsf{S}_i.(\mathsf{B}_k \land b_k > 0) \Rightarrow \mathrm{wlp}.\mathsf{S}_i.X)\\
\equiv\ & \{ \text{ Definición de } P.s_m \ \}\\
& (\mathsf{B}_i \Rightarrow \mathrm{wlp}.\mathsf{S}_i.\neg(\mathsf{B}_k \land b_k > 0))\\
& \Rightarrow (\mathsf{B}_i \land \mathrm{wlp}.\mathsf{S}_i.(\mathsf{B}_k \land b_k > 0) \Rightarrow \mathrm{wlp}.\mathsf{S}_i.X)\\
\equiv\ & \{ \text{ Implicación escrita como disyunción } \}\\
& \neg\mathsf{B}_i \lor \mathrm{wlp}.\mathsf{S}_i.\neg(\mathsf{B}_k \land b_k > 0)\\
& \Rightarrow (\mathsf{B}_i \land \mathrm{wlp}.\mathsf{S}_i.(\mathsf{B}_k \land b_k > 0) \Rightarrow \mathrm{wlp}.\mathsf{S}_i.X)\\
\equiv\ & \{ \text{ Implicación anidada } \}\\
& (\neg\mathsf{B}_i \lor \mathrm{wlp}.\mathsf{S}_i.\neg(\mathsf{B}_k \land b_k > 0)) \land \mathsf{B}_i \land \mathrm{wlp}.\mathsf{S}_i.(\mathsf{B}_k \land b_k > 0)\\
& \Rightarrow \mathrm{wlp}.\mathsf{S}_i.X\\
\equiv\ & \{ \text{ Distributividad y eliminación por negación } \}\\
& \mathsf{B}_i \land \mathrm{wlp}.\mathsf{S}_i.\neg(\mathsf{B}_k \land b_k > 0) \land \mathrm{wlp}.\mathsf{S}_i.(\mathsf{B}_k \land b_k > 0)\\
& \Rightarrow \mathrm{wlp}.\mathsf{S}_i.X
\end{aligned}
$$

$\equiv$ { Conjuntividad de wlp }

$\mathsf{B}_i \wedge \text{wlp}.\mathsf{S}_i.(\neg(\mathsf{B}_k \wedge b_k > 0) \wedge \mathsf{B}_k \wedge b_k > 0)$

$\Rightarrow \text{wlp}.\mathsf{S}_i.X$

$\equiv$ { Lógica }

$\mathsf{B}_i \wedge \text{wlp}.\mathsf{S}_i.\mathsf{false} \Rightarrow \text{wlp}.\mathsf{S}_i.X$

$\equiv$ { Wlp es estricto ya que S_i es total }

$\mathsf{B}_i \wedge \mathsf{false} \Rightarrow \text{wlp}.\mathsf{S}_i.X$

$\equiv$ { Lógica }

true

Por lo tanto, el término conjuntivo en WLP.$\mathcal{T}.X$ correspondiente a la transición, es absorbido por P cuando se calcula el punto fijo. Esto quiere decir que la transición anterior no contribuye a su calculo y la misma puede ser eliminada. Notar que en el caso que $k = i$ (se elimina la guarda que libera procesos del mismo tipo), esta transición es una de las eliminadas en el caso anterior (sin necesidad de utilizar el invariante candidato).

El mismo análisis puede hacerse con la transición de clase 4:

$$(s_i,\ \square\ \text{wlp}.(b_i := b_i - 1; \mathsf{S}_i).(\mathsf{B}_k \wedge b_k > 0) \mapsto b_i := b_i - 1; \mathsf{S}_i,\ s_k)$$

y la componente $P.s_i$ del invariante candidato.

Con los resultados vistos hasta ahora se pueden eliminar $m + 1$ o $m + 2$ transiciones, dependiendo si la guarda a eliminar libera o no un proceso del mismo tipo ($k = i$ o $k \neq i$). A continuación veremos que hay guardas en las sentencias de transiciones que pueden ser simplificadas.

Simplificación de guardas en transiciones

Por último las guardas de las transiciones de clase 3 (desde in_m a out_m) pueden ser simplificadas. Las mismas tienen la forma:

$\mathsf{B}_i \wedge \text{wlp}.\mathsf{S}_i.\langle \forall j\ : 0 \leqslant j < m :\ \neg\mathsf{B}_j\ \vee\ b_j = 0 \rangle$

$\equiv$ { Separación de término, conjuntividad de wlp }

$\mathsf{B}_i \wedge \text{wlp}.\mathsf{S}_i.(\neg\mathsf{B}_i\ \vee\ b_i = 0)$

$\wedge\ \text{wlp}.\mathsf{S}_i.\langle \forall j\ : 0 \leqslant j < m \wedge j \neq i :\ \neg\mathsf{B}_j\ \vee\ b_j = 0 \rangle$

Demostraremos entonces que, bajo la invariancia de φ_{SBD}, el término B_i es más fuerte que wlp.$\mathsf{S}_i.(\neg\mathsf{B}_i\ \vee\ b_i = 0)$, simplificándose así la cuantificación universal:

$\mathsf{B}_i \Rightarrow \text{wlp}.\mathsf{S}_i.(\neg\mathsf{B}_i\ \vee\ b_i = 0)$

$\equiv$ { S_i es determinista }

$\mathsf{B}_i \Rightarrow \text{wlp}.\mathsf{S}_i.\neg\mathsf{B}_i\ \vee\ \text{wlp}.\mathsf{S}_i.(b_i = 0)$

$\equiv$ { El programa S_i no modifica contadores }

$\mathsf{B}_i \Rightarrow \text{wlp}.\mathsf{S}_i.\neg\mathsf{B}_i\ \vee\ b_i = 0$

$\equiv$ { Implicación como disyunción, conmutatividad }

$\neg\mathsf{B}_i\ \vee\ b_i = 0\ \vee\ \text{wlp}.\mathsf{S}_i.\neg\mathsf{B}_i$

$\equiv$ { $\neg\mathsf{B}_i \lor b_i = 0$ es un término conjuntivo del invariante φ_{SBD} }

$\mathsf{true} \lor \text{wlp}.\mathsf{S}_i.\neg\mathsf{B}_i$

$\equiv$ { Lógica }

true

Por lo tanto las transiciones anteriores pueden escribirse como

$$(\, s_m,\ \square\ \mathsf{B}_i \land \text{wlp}.\mathsf{S}_i.\langle \forall j\ : 0 \leqslant j < m \land j \neq i :\ \neg\mathsf{B}_j \lor b_j = 0 \rangle \mapsto \mathsf{S}_i,\ s_m\,)$$

$(0 \leqslant i < m)$ eliminando un término conjuntivo dentro de la cuantificación universal.

4.6.3 Simplificación del invariante candidato

En el caso que se intente eliminar, en un programa S_i, la guarda $\mathsf{B}_i \land b_i > 0$ (se libera un proceso ejecutando el mismo programa), por la ecuación 4.4 (pág. 130), el invariante candidato en la locación s_m estará definido como:

$$P.s_m \ \doteq\ \mathsf{B}_i \Rightarrow \text{wlp}.\mathsf{S}_i.\neg(\mathsf{B}_i \land b_i > 0)\ .$$

Notemos además, que el invariante φ_{SBD} (ecuación 4.3, pág. 125) contiene el término conjuntivo $\neg\mathsf{B}_i \lor b_i = 0$, el cual nos permitirá simplificar el predicado anterior:

$P.s_m$

$\equiv$ { Definición de P }

$\mathsf{B}_i \Rightarrow \text{wlp}.\mathsf{S}_i.\neg(\mathsf{B}_i \land b_i > 0)$

$\equiv$ { Lógica }

$\mathsf{B}_i \Rightarrow \text{wlp}.\mathsf{S}_i.(b_i > 0 \Rightarrow \neg\mathsf{B}_i)$

$\equiv$ { Como S_i es determinista wlp distribuye con la implicación }

$\mathsf{B}_i \Rightarrow (\text{wlp}.\mathsf{S}_i.(b_i > 0) \Rightarrow \text{wlp}.\mathsf{S}_i.\neg\mathsf{B}_i)$

$\equiv$ { S_i no modifica b_i }

$\mathsf{B}_i \Rightarrow (b_i > 0 \Rightarrow \text{wlp}.\mathsf{S}_i.\neg\mathsf{B}_i)$

$\equiv$ { Lógica, aritmética }

$\neg\mathsf{B}_i \lor b_i = 0 \lor b_i < 0 \lor \text{wlp}.\mathsf{S}_i.\neg\mathsf{B}_i)$

$\equiv$ { Suposición $\neg\mathsf{B}_i \lor b_i = 0$ por invariante φ_{SBD} }

$\mathsf{true} \lor b_i < 0 \lor \text{wlp}.\mathsf{S}_i.\neg\mathsf{B}_i)$

$\equiv$ { Lógica }

true

Por lo tanto, al momento de intentar eliminar la guardada que libera procesos del mismo tipo, se puede definir $P.s_m \doteq \mathsf{true}$.

4.6.4 Simplificación de fórmulas

Como ya mencionamos, el cálculo recurrente del transformador WLP en el proceso de búsqueda del punto fijo produce un aumento del tamaño en la representación de los predicados. Este fenómeno conduce al agotamiento de los recursos computacionales al momento de ejecutar el método, lo cual suele demorar su terminación o incluso puede detener su progreso. Para tratar este problema se implementaron diversas estrategias con el fin de reducir el tamaño de las fórmulas manteniendo su equivalencia.

Dentro del método de demostración de k-invariancia por propagación hacia atrás, se agregó la función **simplificar** como se muestra en el programa 4.15 (dentro de la definición del transformador $\mathcal{B}_{\mathcal{T},P}$) que implementa las estrategias de simplificación. Como se muestra, la misma es aplicada sobre los elementos de la cadena calculados almacenados en la variable B_k utilizando además el invariante φ_{SBD}. El mismo es utilizado como suposición de todas las pruebas involucradas en el proceso de simplificación que se describe a continuación.

Programa 4.15 Propagación hacia atrás con simplificación

es_invariante ($P, \Theta : \mathsf{Pred}_{\mathcal{L}}$; $\mathcal{T} : \mathsf{Tran}_{\mathcal{L},\mathcal{S}}$) : Bool

$\mathcal{B}_{\mathcal{T},P} := \langle \lambda X \bullet \textbf{simplificar}.\varphi_{SBD}.(P \cap \text{WLP}.\mathcal{T}.X) \rangle$;

$B_k := \mathsf{True}$;

<u>**do**</u> $\neg(\varphi_{SBD} \vdash \neg(\Theta \subseteq B_k)) \wedge \neg(\varphi_{SBD} \vdash B_k \subseteq \mathcal{B}_{\mathcal{T},P}.B_k)$

$B_k := \mathcal{B}_{\mathcal{T},P}.B_k$

<u>**od**</u>;

return $\varphi_{SBD} \vdash \Theta \subseteq B_k$

Debido a las definición del transformador WLP estas fórmulas resultan libres de cuantificadores, por lo cual los predicados en las locaciones pueden ser transformadas a su forma normal conjuntiva. Si se puede probar (utilizando φ_{SBD} como suposición) que algún término conjuntivo de la fórmula normalizada es más débil que la conjunción de los demás, el mismo es eliminado por absorción. Este proceso es aplicado a todos los términos de la forma normal conjuntiva.

De la misma manera, también fue aplicada la táctica dual sobre cada término conjuntivo anterior: si un subtérmino disyuntivo es más fuerte (suponiendo el invariante φ_{SBD}) que la disyunción de los demás subtérminos, entonces es eliminado. Las pruebas de inclusión entre términos fueron implementadas utilizando el SMT solver CVC3.

Otras simplificaciones fueron aplicadas utilizando el probador de teoremas Isabelle/HOL. Esta herramienta implementa tácticas de simplificación por defecto denominadas “clarify” y “simplify” las cuales fueron utilizadas dentro de la función **simplificar** con el fin de reducir el tamaño de las fórmulas.

Todas las tácticas descriptas fueron implementadas en el lenguaje ML utilizando los probadores externos mencionados. Este desarrollo forma parte del prototipo de software que se describe a continuación.

4.7 Implementación

Con el fin de probar el método descripto sobre diversos problemas de regiones críticas condicionales, se desarrolló un prototipo de software que genera el sistema de transiciones e implementa el método de demostración de invariancia. El software fue escrito en el lenguaje ML, en su implementación *Standard ML of New Jersey* [Sta06], utilizando como procesos externos los probadores de teoremas CVC3 e Isabelle/HOL.

Resumiendo, el prototipo de software implementa las siguientes etapas:

1. A partir de una especificación de un problema particular de regiones críticas condicionales se genera un sistema de transiciones que modela la ejecución de los programas obtenidos mediante la técnica SBD, junto con el invariante φ_{SBD} que asegura la exclusión mutua condicional. La base de este proceso es descripto en la sección 4.4 (pág. 118). Además, se eliminan transiciones y se simplifican las guardas del sistema, como se indica en la sección 4.6.2.

2. Eligiendo una guarda del **if** final se obtiene un invariante candidato P que captura la imposibilidad de ejecución de misma, como se indica en la sección 4.5. Este invariante puede ser simplificado como se muestra en la sección 4.6.3.

3. Finalmente, utilizando la técnica de propagación hacia atrás esquematizada en el programa 4.15 (pág. 137), se verifica si el invariante candidato es un invariante del sistema. Con ello se puede decidir si la guarda del programa puede ser eliminada.

En la figura 4.1 (pág. 139) se muestra un diagrama esquemático del diseño del prototipo donde se señalan los módulos que implementan cada etapa. Las etapas 1 y 2 fueron implementadas en el módulo *Generador de Transiciones* y la etapa restante en el modulo *Verificador de Invariantes*. A continuación describiremos los detalles de implementación de ambos módulos.

4.7.1 Generador de Transiciones

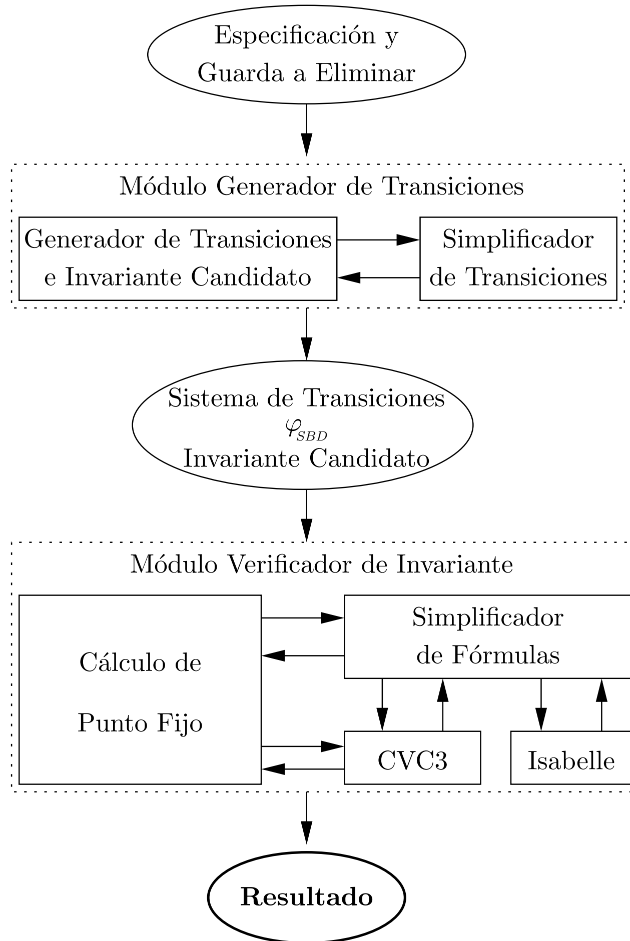
Este módulo genera el sistema de transiciones y el invariante candidato a partir de la especificación de un problema de regiones críticas condicionales y la información del comando guardado final que se desea eliminar. En su forma más general, un problema de regiones críticas condicionales puede ser especificado como:

- un conjunto de estados iniciales θ.

- una secuencia de m programas $\mathsf{S}_0, \cdots, \mathsf{S}_{m-1}$ a ejecutarse en exclusión mutua condicional,

- m condiciones $\mathsf{B}_0, \cdots, \mathsf{B}_{m-1}$ de ejecución para los programas anteriores respectivamente,

Figura 4.1: Diseño.



- un invariante global del sistema I que debe ser mantenido por las ejecuciones de dichos programas[3].

La especificación es tomada en la forma de un archivo de texto, el cual es parseado produciendo diversos tipos de datos ML para su procesamiento. A partir de estos se genera el sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ y el invariante candidato simplificados. También se devuelve como resultado el invariante φ_{SBD}. Los predicados envueltos son representados por un tipo de datos recursivo ML y los predicados sobre el sistema de transiciones son listas de longitud $m+1$ de estos tipos de datos. En tal sentido, el conjunto de locaciones estará representado por el intervalo de números naturales $\mathcal{L} \doteq \{0, \cdots, m\}$.

[3]Las condiciones y el invariante global son redundantes, en el sentido que unas pueden ser obtenidos del otro de forma mecánica o viceversa. Esta previsto agregar esta funcionalidad en versiones futuras del software.

4.7.2 Verificador de Invariantes

Este módulo toma el sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ generado junto con el invariante φ_{SBD} y el invariante candidato, e implementa el método de demostración de k-invariancia esquematizado en el programa 4.15 (pág. 137).

Las pruebas de las implicaciones involucradas en el cálculo de punto fijo (tomando como válido φ_{SBD}) fueron realizadas utilizando CVC3. Este SMT solver admite fórmulas no lineales pero es incompleto en este tipo de teorías. Como se menciona en la sección 4.5, los sistemas de transiciones resultantes suelen ser lineales, por lo que nuestro método produjo resultados positivos. De todas maneras, el prototipo permite utilizar otros demostradores como Isabelle/HOL y ACL2 que poseen estrategias complementarias para resolver fórmulas en este tipo de teorías. Cabe mencionar que en los casos de prueba, la utilización de estos demostradores en reemplazo de CVC3, condujo a una perdida de eficiencia por agotamiento de los recursos computacionales.

Para hacer más eficiente el cálculo del punto fijo realizado por este módulo, incluimos las técnicas de simplificación de las formulas obtenidas en cada paso de este cálculo, como se indica en la sección 4.6. Las mismas fueron implementadas utilizando CVC3 y el probador de teoremas Isabelle/HOL en conjunto con técnicas de simplificación propias codificadas en ML (dentro del segundo módulo) como se indica en la sección 4.6.4. La interfase entre los probadores externos y el prototipo escrito en ML fue resuelta simplemente ejecutando los primeros como procesos independientes desde el segundo. El control de los probadores se realizó accediendo a las entradas y salidas estándar de los procesos que los ejecutan.

4.8 Resultados

El prototipo de software fue probado sobre diversos problemas de la literatura. Este programa fue corrido sobre una computadora con arquitectura PC, procesador de 2GHz. y 2Gb de memoria RAM. A continuación se detallan los resultados obtenidos.

4.8.1 Productor/Consumidor en buffer acotado

Consideremos el ejemplo 4.1 (pág. 111) donde se resuelve el problema de productores y consumidores.

Como primera prueba se corrió el método con invariante φ_{SBD} igual a True e invariante candidato P igual a φ_{SBD}. Este caso fue pensado únicamente como test, sin pretender eliminar ninguna guarda, y solo para comprobar si φ_{SBD} es un invariante del sistema.

Después de efectuar las simplificaciones y eliminaciones del sistema de transiciones, las transiciones generadas fueron las mostradas en las figura 4.2 (pág. 141). Allí las locaciones son representadas por el índice del semáforo, tal cual se representa dentro del prototipo, como se indicó en la sección 4.7.1. Notar que después de realizar las simplificaciones se obtienen dos transiciones menos y se simplificaron las guardas de las transiciones que van de la locación 2 a ella misma (semáforo s_m), como resultado de la aplicación de las simplificaciones descriptas en la sección 4.6.2 (pág. 133).

$$
\begin{aligned}
\mathcal{T} \doteq \{\ & (2, \square\, \neg n < N \mapsto b_0 := b_0 + 1,\ 2)\ , \\
& (2, \square\, \neg n > 0 \mapsto b_1 := b_1 + 1,\ 2)\ , \\
& (2, \square\, n < N \wedge b_1 > 0 \wedge n + 1 > 0 \mapsto n := n + 1,\ 1)\ , \\
& (2, \square\, n > 0 \wedge b_0 > 0 \wedge n - 1 < N \mapsto n := n - 1,\ 0)\ , \\
& (2, \square\, n < N \wedge (b_1 = 0\ \vee\ \neg n + 1 > 0) \mapsto n := n + 1,\ 2)\ , \\
& (2, \square\, n > 0 \wedge (b_0 = 0\ \vee\ \neg n - 1 < N) \mapsto n := n - 1,\ 2)\ , \\
& (0, \square\, b_0 - 1 > 0 \wedge n + 1 < N \mapsto b_0, n := b_0 - 1, n + 1,\ 0)\ , \\
& (0, \square\, b_1 > 0 \wedge n + 1 > 0 \mapsto b_0, n := b_0 - 1, n + 1,\ 1)\ , \\
& (1, \square\, b_0 > 0 \wedge n - 1 < N \mapsto b_1, n := b_1 - 1, n - 1,\ 0)\ , \\
& (1, \square\, b_1 - 1 > 0 \wedge n - 1 > 0 \mapsto b_1, n := b_1 - 1, n - 1,\ 1)\ , \\
& (0, \square\, (b_0 - 1 = 0\ \vee\ \neg n + 1 < N) \wedge (b_1 = 0\ \vee\ \neg n + 1 > 0) \\
& \qquad\qquad \mapsto b_0, n := b_0 - 1, n + 1,\ 2)\ , \\
& (1, \square\, (b_0 = 0\ \vee\ \neg n - 1 < N) \wedge (b_1 - 1 = 0\ \vee\ \neg n - 1 > 0) \\
& \qquad\qquad \mapsto b_1, n := b_1 - 1, n - 1,\ 2)\ \}
\end{aligned}
$$

Figura 4.2: Transiciones de Productor/Consumidor sin eliminación de guardas.

El método converge en una iteración, siendo el punto fijo $\nu.\mathcal{B}_{\mathcal{T},P}$ igual a φ_{SBD}. Esto corrobora el hecho que φ_{SBD} es un invariante inductivo del sistema.

En una segunda prueba, intentamos eliminar el la guarda del procesos productor $n < N \wedge b_0 > 0$. Esta guarda libera otro procesos productor esperando en el semáforo. Las transiciones computadas son mostradas en la figura 4.3 (pág. 142), el invariante candidato es:

$$P \doteq [b_0 - 1 = 0\ \vee\ \neg n + 1 < N, \mathsf{true}, \mathsf{true}]$$

y las configuraciones iniciales son:

$$\Theta \doteq [\mathsf{false}, \mathsf{false}, n = 0 \wedge 0 < N \wedge b0 = 0 \wedge b1 = 0]\ .$$

Después de algo más de un segundo, el método converge en 5 iteraciones devolviendo el siguiente invariante:

$$
\begin{aligned}
\nu\mathcal{B}_{\mathcal{T},P} \doteq\ & [(n + 1 < N \Rightarrow b_0 = 1) \\
& \quad \wedge (0 < b_1 - 1 \Rightarrow 0 < n \Rightarrow b_0 = 2\ \vee\ \neg 0 < b_0 - 1), \\
& \ (0 < b_0 \Rightarrow 0 < b_1 - 1 \Rightarrow b_0 = 1\ \vee\ \neg 0 < n - 1) \\
& \quad \wedge (0 < b_0 \Rightarrow b_0 = 1\ \vee\ \neg n < N), \mathsf{true}]
\end{aligned}
$$

el cual es implicado por Θ. Por lo tanto la guarda puede ser eliminada.

También intentamos eliminar la guarda del consumidor $n > 0 \wedge b_1 > 0$ (libera a otro consumidor) con similares resultados positivos.

Con respecto al resto de las guardas ($n > 0 \wedge b_1 > 0$ en el productor y $n < N \wedge b_0 > 0$ en el consumidor) el método detecta la no satisfabilidad de la fórmula $\Theta \subseteq B_k$ en pocas iteraciones.

$$
\begin{array}{rll}
\mathcal{T} & \doteq \{ & (2,\ \square\, \neg n < N \mapsto b_0 := b_0 + 1,\ 2)\ , \\
& & (2,\ \square\, \neg n > 0 \mapsto b_1 := b_1 + 1,\ 2)\ , \\
& & (2,\ \square\, n < N \wedge b_1 > 0 \wedge n + 1 > 0 \mapsto n := n + 1,\ 1)\ , \\
& & (2,\ \square\, n > 0 \wedge b_0 > 0 \wedge n - 1 < N \mapsto n := n - 1,\ 0)\ , \\
& & (2,\ \square\, n < N \wedge (b_1 = 0\ \vee\ \neg n + 1 > 0) \mapsto n := n + 1,\ 2)\ , \\
& & (2,\ \square\, n > 0 \wedge (b_0 = 0\ \vee\ \neg n - 1 < N) \mapsto n := n - 1,\ 2)\ , \\
& & (0,\ \square\, b_1 > 0 \wedge n + 1 > 0 \mapsto b_0, n := b_0 - 1, n + 1,\ 1)\ , \\
& & (1,\ \square\, b_0 > 0 \wedge n - 1 < N \mapsto b_1, n := b_1 - 1, n - 1,\ 0)\ , \\
& & (1,\ \square\, b_1 - 1 > 0 \wedge n - 1 > 0 \mapsto b_1, n := b_1 - 1, n - 1,\ 1)\ , \\
& & (0,\ \square\, (b_0 - 1 = 0\ \vee\ \neg n + 1 < N) \wedge (b_1 = 0\ \vee\ \neg n + 1 > 0) \\
& & \qquad\qquad \mapsto b_0, n := b_0 - 1, n + 1,\ 2)\ , \\
& & (1,\ \square\, (b_0 = 0\ \vee\ \neg n - 1 < N) \wedge (b_1 - 1 = 0\ \vee\ \neg n - 1 > 0) \\
& & \qquad\qquad \mapsto b_1, n := b_1 - 1, n - 1,\ 2)
\end{array}
$$

Figura 4.3: Transiciones de Productor/Consumidor eliminando guarda.

La tabla 4.1 muestra el detalle de estos resultados. La primer y segunda columna indican el programa con su guarda a eliminar, la tercera la cantidad de transiciones generadas y las siguientes el resultado, la cantidad de iteraciones y el tiempo de cómputo (wall time).

Programa	Guarda	#Trs.	Resultado	Iteraciones	Wall Time
Prod	Prod	11	eliminada	5	1.127”
Prod	Cons	10	no eliminada	3	0.132”
Cons	Prod	10	no eliminada	4	0.415”
Cons	Cons	11	eliminada	5	0.462”

Tabla 4.1: Resultado de Productor/Consumidor

En el ejemplo 4.1 vimos que la formulación original del problema envuelve el uso de dos variables y un arreglo (buffer), mientras que en la especificación que usamos se eliminó la sentencia de acceso al arreglo y se realizó un cambio de coordenadas a una sola variable n. La eliminación del acceso al arreglo puede hacerse de manera automática, ya que esta estructura no modifica las condiciones de sincronización (guardas). La segunda modificación no es simple de realizar en forma automática, por lo cual queda la pregunta si es posible utilizar nuestro método sin ella.

La especificación sin el cambio de coordenadas es:

resource *prod_cons* $(p, d : \textbf{int})$ $\theta \doteq p = 0 \wedge d = 0 \wedge N > 0$	
Prod: **region** *prod_cons* $\square\, p - d < N \mapsto p{:=}p + 1$ **end**	Cons: **region** *prod_cons* $\square\, p - d > 0 \mapsto d{:=}d + 1$ **end**

Los resultados para esta nueva formulación del problema se muestran en la tabla 4.2.

Programa	Guarda	#Trs.	Resultado	Iteraciones	Wall Time
Prod	Prod	11	eliminada	5	1.476”
Prod	Cons	10	no eliminada	3	0.195”
Cons	Prod	10	no eliminada	4	0.630”
Cons	Cons	11	eliminada	5	0.656”

Tabla 4.2: Resultado de Productor/Consumidor con dos variables

Como se puede ver, se encontraron todas las soluciones en pocos segundos.

4.8.2 Contraejemplo Schmid

Este caso de test es el contraejemplo al método de Schmid [Sch76] desarrollado en la sección 4.3 (pág. 117). El mismo intenta resolver el problema de productores y consumidores modificado, donde se agrega un proceso que consume únicamente si el buffer tiene un elemento. En aquella sección vimos que el método de Schmid no lo resuelve correctamente.

En este ejemplo hay 9 guardas posibles de eliminar y la cantidad de transiciones es mayor que en el ejemplo anterior (entre 22 y 23 transiciones según la guarda). El resultado de este problema se muestra en la tabla 4.3.

Programa	Guarda	#Trs.	Resultado	Iteraciones	Wall Time
Prod	Prod	23	no eliminada	9	8.592”
Prod	Cons1	22	no eliminada	3	0.273”
Prod	Cons	22	no eliminada	3	0.264”
Cons1	Prod	22	no eliminada	4	1.114”
Cons1	Cons1	23	eliminada	1	0.015”
Cons1	Cons	22	eliminada	1	0.018”
Cons	Prod	22	no eliminada	4	0.622”
Cons	Cons1	22	no eliminada	5	5.963”
Cons	Cons	23	eliminada	5	0.720”

Tabla 4.3: Resultado de Productor/Consumidor modificado

En todos los casos el método pudo decidir la eliminación o no de la guarda. En los casos que el método de Schmid falla, nuestro método encuentra las

soluciones correctas. Notar que justamente el caso donde se demuestra que un productor puede liberar a otro productor (caso contraejemplo en sección 4.3) es donde se tarda mayor tiempo.

4.8.3 Semáforos generales

Este es el problema clásico de implementación de semáforos generales con semáforos binarios [Dij68].
Su especificación es:

resource *semGen* (x : **int**) $\theta \doteq x = 0 \quad I \doteq x \geqslant 0$	
P: **region** *semGen* □ $x > 0 \mapsto x := x - 1$ **end**	V: **region** *semGen* □ true $\mapsto x := x + 1$ **end**

La posibilidad de eliminación de las cuatro guardas involucradas es decidida correctamente en menos de un segundo. El resultado puede verse en la tabla 4.4.

Programa	Guarda	#Trs.	Resultado	Iteraciones	Wall Time
P	P	11	eliminada	2	0.705”
P	V	10	eliminada	2	0.029”
V	P	10	no eliminada	3	0.083”
V	V	11	eliminada	1	0.004”

Tabla 4.4: Resultado de Semáforos generales

En el problema solo la guarda $x > 0 \wedge b_0 > 0$ en el programa V no puede ser eliminada. La misma corresponde a las situación donde un proceso ejecutando V libera a otro ejecutando P.

4.8.4 Lectores y escritores

El problema de lectores y escritores es uno de los ejemplos utilizados por E.W. Dijkstra para presentar la técnica SBD [Dij79]. Su especificación es:

resource *RW* (r, w : **int**) $\theta \doteq w = 0 \wedge r = 0 \quad I \doteq r \geqslant 0 \wedge (w = 0 \vee (w = 1 \wedge r = 0))$	
Write_in: **region** *RW* □ $w = 0 \wedge r = 0 \mapsto w := w + 1$ **end**; write; Write_out: **region** *RW* □ true $\mapsto w := w - 1$ **end**	Read_in: **region** *RW* □ $w = 0 \mapsto r := r + 1$ **end**; read; Read_out: **region** *RW* □ true $\mapsto r := r - 1$ **end**

Programa	Guarda	#Trs.	Resultado	It.	W. T.
Write_in	Write_in	39	eliminada	1	0.007”
Write_in	Write_out	38	eliminada	2	0.047”
Write_in	Read_in	38	eliminada	1	0.007”
Write_in	Read_out	38	eliminada	2	0.052”
Write_out	Write_in	38	no eliminada	4	0.300”
Write_out	Write_out	39	eliminada	1	0.005”
Write_out	Read_in	38	no eliminada	4	0.238”
Write_out	Read_out	38	eliminada	1	0.006”
Read_in	Write_in	38	eliminada	1	0.008”
Read_in	Write_out	38	eliminada	2	0.045”
Read_in	Read_in	39	no eliminada	6	0.396”
Read_in	Read_out	38	eliminada	2	0.044”
Read_out	Write_in	38	no eliminada	4	0.620”
Read_out	Write_out	38	eliminada	1	0.006”
Read_out	Read_in	38	eliminada	1	0.008”
Read_out	Read_out	39	eliminada	1	0.006”

Tabla 4.5: Resultado de Lectores y escritores

El resultado de la aplicación del método es mostrado en la tabla 4.5. Como se puede observar, el método arriba a la solución correcta: a la salida de la escritura se puede activar una lectura o escritura, a la entrada de una lectura se puede activar solo nuevas lecturas y a la salida de una lectura se puede activar solo una escritura.

Cabe destacar que en este ejemplo se ve claramente la abstracción realizada al modelar el problema con sistema de transiciones (ver nota en pág. 124). Sin embargo el método obtuvo la solución correcta. Esto es debido a que el invariante del problema, provisto de forma manual, implica que w (cantidad de escritores) y r (cantidad de lectores) son mayores o iguales a cero, lo cual denota la imposibilidad ejecución de las regiones críticas de salida si no hay procesos que ejectaron las de entrada.

4.8.5 Productor/Consumidor goloso

Este caso de test es una versión modificada del problema de productores y consumidores donde el último consume dos elementos en vez de uno. Como veremos, el hecho rompe la simetría entre las dos clases de procesos.
La especificación de este problema de regiones críticas condicionales es:

resource *prod_cons* (n : **int**) $\theta \doteq n = 0 \wedge N > 1 \qquad I \doteq 0 \leqslant n \leqslant N$	
Prod: **region** *prod_cons* □ $n < N \mapsto n := n + 1$ **end**	Cons: **region** *prod_cons* □ $n > 1 \mapsto n := n - 2$ **end**

Para el caso de la guarda del consumidor liberando a otro proceso del mismo tipo, el sistema de transiciones generado tiene 11 transiciones y el punto fijo es alcanzado en casi 2 segundos, obteniéndose el siguiente invariante:

$$
\begin{array}{l}
\nu\mathcal{B}_{\mathcal{T},P} \ \doteq \\
\quad [(0 < b_1 \Rightarrow b_1 = 1 \ \vee\ \neg 3 <= n) \\
\quad\ \wedge (0 < b_1 \Rightarrow 0 < -2 + b_0 \Rightarrow 2 + n < N \Rightarrow b_1 = 1 \ \vee\ \neg 1 <= n) \\
\quad\ \wedge (3 + n < N \Rightarrow 0 < b_1 \Rightarrow b_1 = 1 \ \vee\ \neg 0 < -3 + b_0) \\
\quad\ \wedge (2 <= n \Rightarrow 0 < b_1 \Rightarrow 0 < b_0 - 1 \Rightarrow b_1 = 1 \ \vee\ \neg n + 1 < N), \\
\quad (3 < n \Rightarrow b_1 = 1) \wedge (0 < b_1 - 1 \Rightarrow 0 < -3 + b_0 \Rightarrow b_1 = 2 \ \vee\ \neg 1 + n < N) \\
\quad\ \wedge (0 < -2 + b_0 \Rightarrow n < N \Rightarrow 3 <= n \Rightarrow b_1 = 2 \ \vee\ \neg 0 < b_1 - 1), \\
\quad \mathsf{true}]
\end{array}
$$

El punto fijo es implicado por el conjunto de estados iniciales, por lo que la guarda puede ser eliminada.

Para las otras guardas el método detecta la no satisfabilidad de la fórmula $\Theta \subseteq B_k$. Los resultados son mostrados en la tabla 4.6.

Programa	Guarda	#Trs.	Resultado	Iteraciones	Wall Time
Prod	Prod	11	no eliminada	7	1.781”
Prod	Cons	10	no eliminada	4	1.082”
Cons	Prod	10	no eliminada	5	1.057”
Cons	Cons	11	eliminada	7	1.539”

Tabla 4.6: Resultado de Productor/Consumidor goloso

Como puede verse, en este problema la única guarda que puede ser eliminada es $n > 1 \wedge b_1 > 0$ dentro del programa consumidor. La misma libera otro consumidor esperando en el semáforo. La guarda del productor para liberar otro proceso del mismo tipo no puede ser eliminada ya que un consumidor puede liberar dos lugares en el buffer, permitiendo que dos productores los ocupen, uno después del otro.

El problema fue alterado variando la cantidad de elementos que consume el consumidor. Con ello se crearon una serie distinta de problemas de regiones críticas condicionales. Por ejemplo, para el problema en que se consumen 7 elementos su especificación será:

<table>
<tr><td colspan="2"><u>resource</u> prod_cons (n : int)
$\theta \ \doteq\ n = 0 \wedge N > 6 \qquad I \ \doteq\ 0 \leqslant n \leqslant N$</td></tr>
<tr><td>Prod: <u>region</u> prod_cons
$\square\ n < N \mapsto n := n + 1$
<u>end</u></td><td>Cons: <u>region</u> prod_cons
$\square\ n > 6 \mapsto n := n - 7$
<u>end</u></td></tr>
</table>

En la tabla 4.7 (pág. 147) se muestra el tiempo total que toma el método para encontrar la posibilidad de eliminación de todas las guardas, la cantidad de iteraciones máximas y mínimas para decidir el resultado en cada guarda a

Consumidos	# Iter. min–max	Tam. invariante	W. T. Total
2	4–7	147	5.751”
3	5–9	227	13.211”
4	6–11	307	29.187”
5	7–13	387	61.161”
6	8–15	467	125.675”
7	9–17	547	257.657”

Tabla 4.7: Resultado de Productor/Consumidor goloso para distinta cantidad de elementos consumidos

eliminar y la cantidad de símbolos del invariante encontrado en la eliminación de la guarda final del consumidor.

En estos resultados se puede apreciar que la cantidad de iteraciones y el tamaño de las fórmulas tienen un comportamiento lineal con respecto a la cantidad de elementos consumidos, mientras que el tiempo de cómputo crece de manera exponencial (se multiplica por dos). Este último fenómeno es producido por la complejidad exponencial de los SMT solvers empleados en el cálculo de punto fijo: aunque las fórmulas intermedias crezcan de forma lineal la complejidad del solver es exponencial con respecto a este tamaño. Este crecimiento lineal en la cantidad de iteraciones inspiró la elaboración del próximo ejemplo.

4.8.6 Productor/Consumidor goloso M

En los ejemplos anteriores el método pudo decidir la eliminación de todas las guardas. Como demostramos en la sección 2.3.4 (pág. 60) existe casos donde el método de propagación hacia atrás no puede encontrar el punto fijo debido a que la cadena calculado es infinita. En la sección 4.5 mencionamos esto como una de las causas de incompletitud de nuestro método.

Por otro parte, en el ejemplo anterior vimos que si se aumenta la cantidad de elementos consumido, la cantidad de iteraciones para alcanzar el punto fijo se comporta de forma lineal con respecto a la magnitud anterior. Esto sugiere que si se reemplaza la cantidad de elementos consumidos por una constante no determinada, estaríamos en presencia de un caso de incompletitud.

Utilizando una constante M para representar la cantidad consumida, la especificación del problema de esta índole será:

<table>
<tr><td colspan="2"><u>resource</u> prod_cons (n : int)
$\theta \doteq n = 0 \wedge N \geqslant M \wedge M > 0 \qquad I \doteq 0 \leqslant n \leqslant N$</td></tr>
<tr><td>Prod: <u>region</u> prod_cons
$\square\, n < N \mapsto n := n + 1$
<u>end</u></td><td>Cons: <u>region</u> prod_cons
$\square\, n \geqslant M \mapsto n := n - M$
<u>end</u></td></tr>
</table>

Para este problema nuestro método no puede decidir la eliminación de la última guarda del consumidor (donde se libera un proceso de su mismo tipo) ya que el punto fijo no converge dentro del máximo de iteraciones establecido.

Para las demás guardas, el método detecta la no posibilidad de eliminación en pocos segundos. Los resultados pueden verse en la tabla 4.8.

Programa	Guarda	#Trs.	Resultado	Iteraciones	Wall Time
Prod	Prod	11	no eliminada	7	19.90”
Prod	Cons	10	no eliminada	3	2.36”
Cons	Prod	10	no eliminada	4	4.53”
Cons	Cons	11	no converge	60	2807.051”

Tabla 4.8: Resultado de Productor/Consumidor goloso M.

Para corroborar la hipótesis que la cadena de invariantes candidatos es infinita, se analizaron los elaborados por el algoritmo (el software provee la posibilidad de mostrar estos resultados intermedios). Un análisis profundo de los mismos es dificultoso ya que estas fórmulas son muy grandes (alrededor de 3000 símbolos cada una). De todas maneras se pudo observar que en cada paso de iteración k los invariantes candidatos muestran términos de la forma $M \leqslant k \Rightarrow E$ con E un predicado, lo cual indica que en cada iteración el método realiza una búsqueda en el espacio de estados para cada valor posible de la variable M. Esta búsqueda no termina ya que el valor M no es definido como sucede en los anteriores problemas de Productor/Consumidor.

4.9 Conclusiones y trabajos futuros

Las regiones críticas condicionales son construcciones de alto nivel para la programación concurrente. Esta característica las hace una herramienta interesante para la construcción de software sin errores, pensándolo dentro de un contexto donde los mismos han sido más la regla que la excepción [Gib94, Cre05, Lee06].

El principal problema que atentó contra el uso de estas construcciones fue la falta de implementaciones eficientes. Actualmente, gracias a la madurez que han alcanzado los probadores se teoremas, se puede pensar en utilizar estas herramientas para solucionar el problema.

En este trabajo se presenta un método para optimizar de forma totalmente automática implementaciones de regiones críticas condicionales realizadas mediante la técnica SBD. El método fue implementado en un prototipo de software que utiliza diversos probadores de teoremas externos. El mismo sirvió para probar la técnica sobre diversos problemas clásicos de la literatura obteniéndose resultados positivos. Estos fueron obtenidos en pocos segundos, lo cual muestra la viabilidad de utilización de la técnica en compiladores y generadores de código.

El método presentado posee tres partes mutuamente relacionadas: el modelado de los implementaciones SBD con sistemas de transiciones, la prueba de la optimizaciones mediante el cálculo del mayor punto fijo y las simplificaciones, tanto del sistema de transiciones como de las fórmulas intermedias en el cálculo anterior. Esto último fue esencial al momento de poder encontrar las mejoras en tiempos razonables.

El análisis sobre nuestro método muestra que es incompleto, en el sentido que puede no encontrar todas las optimizaciones posibles. Una de las causas de incompletitud es la no convergencia en el cálculo del punto fijo. Si bien no disponemos de una caracterización de los problemas para los cuales el método converge, este funciona correctamente en los ejemplos tomados de la literatura.

En su estado actual, la técnica podría ser aplicada para la optimización de programas simplemente limitando el número de iteraciones en el cálculo del punto fijo o mediante algún límite de tiempo. Si bien esto puede dejar fuera programas convergentes, el fenómeno no necesariamente puede ser resuelto en general, dado que el algoritmo tiene que chequear también implicaciones en la aritmética, lo cual no siempre es decidible.

El método propuesto podría ser mejorado incorporando técnicas de interpretación abstracta (capítulo 3) sobre distintos dominios, de forma tal de acelerar la convergencia con nuevos invariantes. Hasta el momento se intentó hacerlo sobre el dominio de poliedros convexos, pero solo se encontraron invariantes más débiles que los producidos automáticamente por la técnica SBD, lo cual no permite realizar nuevas simplificaciones. Como veremos en el próximo capítulo, el empleo de esta técnica fue esencial para resolver el problema allí propuesto. Queda pendiente para trabajos futuros la utilización de otros dominios de interpretación abstracta como abstracción por predicados con refinamiento por contraejemplos [CGJ+00, DD02]. También sería interesante estudiar la aplicabilidad de alguna técnica de narrowing sobre el cálculo del máximo punto fijo al estilo de la presentada en [BW10].

Inicialmente este trabajo fue planteado en función de un objetivo general más ambicioso: como utilizar los probadores de teoremas no solo para la verificación de programas si no para ayudar en su construcción. Aunque el trabajo aquí presentado va en esa dirección (no se verifican programas si no que se los modifica para mejorar su eficiencia) queda pendiente poder utilizar estas técnicas de generación de invariantes para guiar el desarrollo de programas concurrentes. Un trabajo futuro es el diseño de una herramienta que funcione a la manera de un laboratorio de invariantes para guiar la construcción de programas. Este objetivo está inspirado en la metodología propuesta por E. W. Dijkstra sobre el tema. Lamentablemente las técnicas asercionales allí propuestas no han tenido gran aceptación debido a las dificultades de las pruebas formales que involucran. Creemos que con el desarrollo actual de los probadores de teoremas, es posible encarar metodologías basadas en estos trabajos donde se facilite las pruebas formales y las búsquedas de invariantes. Para ello se deberán crear interfaces de programación que utilicen probadores externos y ayuden a la visualización de sus resultados dentro del contexto de los programas. Este tipo de trabajos podría dar lugar a una verdadera ciencia de la computación experimental, con experimentos mucho más ricos e informativos que los “casos de test”. Como ya mostró convincentemente Dijkstra, las propiedades de los (quizá aún incompletos) programas son una buena guía para la construcción de los mismos. Lo que estas herramientas habilitarían, es la posibilidad práctica de experimentar con estas propiedades.

Capítulo 5

Implementación de monitores con señalamiento automático

En el capítulo anterior presentamos la técnica SBD la cual permite implementar con semáforos binarios las construcciones concurrentes de alto nivel denominadas regiones críticas condicionales. Vimos también que al aplicarla a problemas particulares, resulta por lo general en programas que son plausibles de ser mejorados en su eficiencia. La dificultad para encontrar estas mejoras en forma manual hace que, a pesar de sus beneficios, estas construcciones no sean ampliamente utilizadas en el desarrollo de programas concurrentes.

A partir de esta problemática desarrollamos un método para encontrar estas mejoras de manera automática, utilizando probadores de teoremas externos. De forma general, aquel trabajo puede ser visto como la presentación de una metodología mediante la cual se pueden implementar, de manera eficiente, construcciones de alto nivel (regiones críticas condicionales) con operaciones de bajo nivel (semáforos), donde estas últimas son generalmente brindadas en los lenguajes de programación concurrente.

El objetivo del trabajo que se abordará en este capítulo corre en un mismo sentido. Aquí presentaremos una metodología para implementar, de manera eficiente y automática, las construcciones concurrentes de alto nivel denominadas *monitores con señalamiento automático* a partir de *monitores con señalamiento explícito* , donde estas últimas son generalmente implementadas en los lenguajes de programación. Del mismo modo que en el capítulo anterior, existen técnicas para realizar estas implementaciones, pero por lo general sus resultados admiten simplificaciones que mejoran su eficiencia. En este trabajo presentaremos una metodología para encontrar estas mejoras de manera automática con un enfoque similar al desarrollado en el capítulo anterior, modelando el comportamiento de los monitores con sistemas de transiciones. Nos concentraremos en la eliminación de señalamientos superfluos, aunque la metodología puede extenderse a otros tipos de mejoras. Además, veremos que el problema actual es más complejo que el anterior por lo que para encontrarlas incluiremos las técnicas de interpretación abstracta en el dominio de poliedros convexos desarrolladas en el capítulo 3.

En las primeras secciones del capítulo presentaremos las categorías de monitores utilizadas en este trabajo y los antecedentes del problema a tratar. En la sección 5.3 veremos una técnica general para implementar monitores de señalamiento automático con monitores de señalamiento explícito. En las secciones 5.4 y 5.5 mostraremos como mejorar estas implementaciones modelando el comportamiento de aquellas implementaciones con sistemas de transiciones y aplicando las técnicas desarrolladas en los capítulos 2 y 3. En las secciones 5.6 y 5.7 veremos algunas mejoras al método desarrollado y su implementación. Finalmente, en las secciones 5.8 y 5.9 mostraremos los resultados de la aplicación del método y las conclusiones del trabajo respectivamente.

5.1 Monitores

En la sección 4.1 del capítulo anterior mencionamos las regiones críticas condicionales como una evolución de las esperas condicionales. Como vimos en aquella ocasión, esas construcciones resolvían algunos problemas en la implementación de las primeras, relacionados a la libertad del contexto donde aparecen las variables de las condiciones en las operaciones **await**. Esta libertad hacía necesaria la utilización de bucles *busy wait* para chequear aquellas condiciones lo cual produce una gran perdida de eficiencia. La estructura de declaración de datos compartidos que introducen las regiones críticas condicionales soluciona este problema: la técnica SBD las implementa utilizando operaciones bloqueantes (semáforos) y no bucles *busy wait*.

De todas maneras, aunque las regiones críticas condicionales fueron un paso importante en la creación de construcciones concurrentes de alto nivel, de manera temprana fueron identificadas algunas de sus deficiencias. En [Han73a] se observa que estas construcciones pueden aparecer en cualquier lugar de los programas, lo que dificulta hacer un seguimiento de la forma en que los datos compartidos son manipulados por los procesos concurrentes. En el mismo artículo se sugiere como solución la utilización de construcciones sintácticas que combinen los datos compartidos con las operaciones sobre los mismos. A estas construcciones las llama monitores. Cabe mencionar que anteriormente en [Han72] el mismo autor propuso extender las regiones críticas condicionales permitiendo que su condición pueda aparecer en cualquier punto de la región crítica (no solamente en su comienzo) lo cual acercó aún más estas construcciones a los monitores actuales.

Siguiendo estas ideas, en [Hoa74] también se menciona la necesidad de unificar datos compartidos con regiones críticas proponiendo una notación inspirada en el sistema de clases de SIMULA67. Pero la modificación más novedosa reside en el agregado de operaciones de señalización explicitas que funcionan para que los procesos comuniquen la liberación del recurso compartido, eliminando de esta forma los predicados en las operaciones bloqueantes. Los monitores con este agregado son los implementados en la mayoría de los lenguajes de programación y serán los primeros que analizaremos. Cabe agregar que en el mismo artículo se menciona que sin esta modificación los monitores son más simples de usar, aunque se dificulta su implementación.

De manera general, los monitores son construcciones brindadas por los lenguajes de programación concurrente que permiten exclusión mutua al acceso de datos compartidos y sincronización entre los procesos que los acceden. Como ya

adelantamos, distintos mecanismos de sincronización han sido propuestos, a partir de los cuales puede hacerse una clasificación de distintos tipos de monitores. En este sentido, los monitores pueden ser divididos en dos grandes categorías: monitores con señalamiento explícito y monitores con señalamiento automático. A continuación profundizaremos sobre esta clasificación, haciendo hincapié en los monitores utilizados en este trabajo.

5.1.1 Señalamiento explícito

Los monitores son construcciones de los lenguajes de programación que permiten organizar el acceso de procesos a recursos compartidos de forma exclusiva. Esta organización se logra mediante mecanismos de sincronización y comunicación que estas construcciones proveen. Un monitor consiste de ciertas variables, que representan los recursos, y un conjunto de procedimientos que operan sobre los mismos. Estos datos pueden ser manipulados directamente solo por aquellos procedimientos. De esta manera, los procesos externos pueden accederlos únicamente de forma indirecta, invocando aquellos procedimientos. Además, como solo se permite la ejecución de un procedimiento al mismo tiempo, la exclusión mutua entre los accesos es garantizada.

Un esquema sintáctico de la declaración de un monitor como una colección de procedimientos es el siguiente:

monitor *Nombre_monitor*
var $v_1 : T_1$
⋮
var $v_m : T_m$
  procedure $op_1(parámetros_1)$
    S_1
  end
  ⋮
  procedure $op_n(parámetros_n)$
    S_n
  end
end

donde $v_1, \cdots, v_m$ son las variables del monitor y $op_1, \cdots, op_n$ son los nombres de los procedimientos con su declaración de parámetros. De esta manera, las invocaciones a los procedimientos del monitor tienen la forma:

$Nombre_monitor.op_i(argumentos)$

Estos procedimientos implementan las operaciones sobre los recursos, representados por las variables del monitor, mediante sus programas $S_1, \cdots, S_n$ respectivos. Los mismos pueden contener variables locales. La ejecución de estos programas se realiza en exclusión mutua.

Diremos entonces que al momento de la ejecución de un procedimiento el monitor permanece en *estado cerrado*. Dentro del mismo, la ejecución de cualquier otro proceso que intente invocar un procedimiento permanecerá detenida.

Llamaremos *proceso activo* al único proceso posible ejecutando un procedimiento en un momento dado. Un monitor en estado cerrado pasará a *estado abierto* cuando el proceso activo libere la ejecución del procedimiento. En esta situación, podrá desbloquearse cualquier otro proceso cuya ejecución haya sido detenida.

Para la resolución de algunos problemas es suficiente la exclusión mutua explicada anteriormente, ya que solo se necesita el acceso serializado al recurso. En otros problemas se hace necesario sincronizar diferentes procesos mientras utilizan el monitor. Es por ello que los monitores proveen la posibilidad de declaración de variables especiales denominadas *variables de condición* (de tipo **cond**) junto con las operaciones **signal** y **wait** sobre las mismas. En el caso que un proceso activo ejecute una operación **wait** sobre una variable de condición, el monitor pasará del estado cerrado al abierto. Al mismo tiempo, el proceso quedará bloqueado en una *cola de procesos* identificada a la variable de condición. En el caso que un proceso activo ejecute una operación **signal** sobre una variable y haya algún proceso en la cola de la misma, este último pasará a otra cola de procesos en espera a ser liberados cuando el monitor se encuentre abierto. El proceso activo, después de la ejecución de la operación **signal**, continuará su ejecución. A partir de esta descripción informal del comportamiento del sistema, puede verse que en la implementación de un monitor entran en juego varias colas de procesos: una cola e de procesos que intentaron invocar un procedimiento mientras el monitor estuvo cerrado y, por cada variable de condición c, una cola w_c para procesos que ejecutaron la operación $\mathbf{wait}.c$ y una cola s_c para procesos que esperaron en una cola w_c y fueron señalizados.

La operación **signal** sobre una variable de condición c puede especificarse como:

$$\begin{array}{rcl} \mathbf{signal}.c & \doteq & \underline{\mathbf{if}}\ |w_c| > 0 \mapsto s_c.\mathbf{enque}(w_c.\mathbf{deque}) \\ & & \,\square\ \ |w_c| = 0 \mapsto \mathbf{skip} \\ & & \underline{\mathbf{fi}} \end{array} \tag{5.1}$$

donde $|w_c|$ es el tamaño de la cola w_c. Esta especificación expresa que, si hay al menos un proceso esperando en la cola w_c (ejecutó una operación $\mathbf{wait}.c$) entonces se lo elimina de la misma y se lo deposita en la cola de señalizados correspondiente. En el caso que no haya procesos esperando, la señalización se pierde. Después de realizarse la operación, el proceso activo seguirá su ejecución normal. Cabe aclarar que este último comportamiento es propio de los monitores denominados *Wait and Notify* los cuales serán los utilizados en este trabajo. Existen otras propuestas de monitores donde un proceso que ejecuta la operación **signal** queda bloqueado (deja de ser activo) permitiendo que otros procesos en las colas de esperas, continúen la ejecución. Estos monitores son denominados de manera genérica *Signal and Wait*. En [BFC95] se hace una clasificación y especificación detallada de los distintos tipos de monitores.

Al momento de ejecutarse un operación **wait** dentro de un procedimiento, el proceso activo deja de serlo, comenzando un nuevo hilo de ejecución por parte de algún proceso señalizado (en la cola s_c para alguna variable de condición c) o de algún procesos intentando la ejecución de algún procedimiento cuando el monitor estuvo cerrado. Este cambio en el contexto de ejecución es realizado por un *scheduler* asociado al monitor, el cual será activado siempre que este pase del estado cerrado al abierto. Teniendo en mente el scheduler, una posible

especificación de la operación **wait** será:

$$\textbf{wait}.c \doteq w_c.\textbf{enque}(\mathit{self})\ ;\ \textbf{schedule} \tag{5.2}$$

donde *self* hace referencia al proceso activo que ejecuta esta operación.

El scheduler elige el próximo proceso activo entre los señalizados y los esperando por invocar un procedimiento. Si el monitor tiene definidas m variables de condición $c_1, \cdots, c_m$, entonces el scheduler puede ser especificado como:

$$\begin{aligned}\textbf{schedule} \doteq\ & \underline{\textbf{if}}\ |e| > 0 \mapsto \mathit{nextActive} := e.\textbf{deque} \\ & \square\ |s_{c_1}| > 0 \mapsto \mathit{nextActive} := s_{c_1}.\textbf{deque} \\ & \quad\vdots \qquad\qquad\qquad \vdots \\ & \square\ |s_{c_m}| > 0 \mapsto \mathit{nextActive} := s_{c_m}.\textbf{deque} \\ & \underline{\textbf{fi}}; \\ & \textbf{run}.\mathit{nextActive}\end{aligned} \tag{5.3}$$

donde la llamada a la función **run** representa el cambio de hilo de ejecución al nuevo proceso activo.

Finalmente, a la salida de un procedimiento, un proceso activo deja al monitor abierto, permitiendo al scheduler habilitar el comienzo de ejecución de algún proceso en espera. Si representamos con la sentencia **return** la salida de un procedimiento, su especificación será simplemente la llamada al scheduler:

$$\textbf{return} \doteq \textbf{schedule} \tag{5.4}$$

Notemos que según la definición del scheduler, este elige de forma no determinista el próximo proceso activo entre los señalizados y los que están esperando invocar un procedimiento. En este sentido, también existen distintas propuestas de monitores que incluyen prioridades entre estas colas. El scheduler especificado, junto a la estrategia de no bloqueo en operaciones **signal**, identifica a los monitores *Wait and Notify* también denominados *No Priority Non-Blocking* (*NPNB*). Otro tipo de monitor que se propone en la literatura, son aquellos que, además de bloquear el proceso activo al ejecutar una operación **signal**, tienen prioridad los procesos señalizados esperando en una cola de condición (colas s_c) sobre los procesos nuevos esperando activar un procedimiento (cola e). Este tipo de monitor es denominado *Signal and Urgent Wait* o *Priority Blocking* (*PB*). Una clasificación completa de los distintos tipos de monitores es detallada en [BFC95].

Otro comportamiento no determinista que presenta el anterior scheduler es la elección entre las colas de señalizados: habiendo procesos de este tipo en distintas colas, el scheduler elije uno de forma no determinista. Este comportamiento es propio de los monitores NPNB como se especifica en [BFC95].

Los monitores NPNB son los más frecuentemente implementados en los lenguajes de programación. Ejemplo de los mismos aparecen en los lenguajes Java, Modula, Mesa, y en la librería del lenguaje C denominada PThreads [BH05]. Aunque esta clase de monitores son ampliamente utilizados, presentan graves inconvenientes al momento de su programación, en comparación con otros tipos de monitores. Por ejemplo, en un monitor *Signal and Urgent Wait*, cuando una variable de condición es señalizada, el control es transferido directamente a un proceso esperando por cierta condición sobre el estado. Esto permite establecer

protocolos de comunicación entre el proceso señalizador y el señalizado a través de las variables de programa del monitor. En contraste, una operación **signal** en un monitor NPNB sirve solo como aviso, a un proceso despertando de una espera, que cierta condición fue satisfecha en el pasado (al momento de ejecutarse el **signal**). Como resultado, la condición por la cual un proceso en espera fue señalizado, puede no cumplirse al momento en que retome la ejecución, haciendo dificultosa la implementación del protocolo. Este problema es causa de muchos errores de programación y ha derivado en un estilo de codificación denominado *signal as hint* [LR80]: después de un **wait** el nuevo proceso activo debe chequear el cumplimiento de la condición por la que fue señalizado. En el caso que no se cumpla, el proceso debe ponerse otra vez en espera (se inserta el **wait** dentro de un bucle). Este estilo de programación suele producir pérdidas de eficiencia debido a los cambios de contexto no necesarios. Como veremos más adelante, este trabajo intenta resolver el mismo problema.

Ejemplo 5.1 (Monitor Productor/Consumidor)
A manera de ejemplo de uso de este tipo de monitores, en el programa 5.1 mostramos una implementación del problema de productores y consumidores presentado en el capítulo 4 (ejemplo 4.1, pág. 111).

Programa 5.1 Monitor Productor/Consumidor señalamiento explícito

```
monitor Productor_Consumidor
  var p, d : int
  var buf : array[0, N) of T
  var c_prod, c_cons : cond
                θ ≐ p = 0 ∧ d = 0 ∧ N > 0

  procedure Prod(x : T)
    if  p − d < N ↦ skip
    ▯ ¬p − d < N ↦
      signal.c_cons;
      wait.c_prod;
      do ¬p − d < N ↦
        wait.c_prod
      od
    fi;
    buf.(p mod N) := x;
    p := p + 1;
    signal.c_prod;
    signal.c_cons
  end

  procedure Cons(var y : T)
    if  p − d > 0 ↦ skip
    ▯ ¬p − d > 0 ↦
      signal.c_prod;
      wait.c_cons;
      do ¬p − d > 0 ↦
        wait.c_cons
      od
    fi;
    y := buf.(p mod N);
    d := d − 1;
    signal.c_prod;
    signal.c_prod
  end
```

La implementación utiliza el estilo de código mencionado anteriormente: las condiciones para producir o consumir un elemento se asocian a variables de condición (c_{prod} y c_{cons}). Si la condición no se cumple, se señaliza a un proceso de otro tipo para que la satisfaga y se espera en una operación **wait** por una señal. En el caso que el proceso despierte de esta operación, se debe volver a

chequear la condición. Esto se logra poniendo la operación dentro de un bucle. A la salida de un proceso se señaliza a otros esperando por sus condiciones.

Esta solución del problema admite ciertas mejoras en su eficiencia. Como se indica en [And91, cap. 6] para este caso en particular, se puede demostrar que los procesos no necesitan señalizar antes de ejecutar una operación **wait**. Además, de forma análoga a la resolución del problema con SBD, las señalizaciones a la salida no son todas necesarias: un productor no necesita señalizar a otro proceso de su misma clase y lo mismo sucede con los consumidores. En [BFC95, sección 4.2.3] se muestras las condiciones de pruebas necesarias para corroborar estas simplificaciones, las cuales involucran la búsqueda de un invariante que se cumpla al momento de las ejecuciones de las operaciones **signal** y **wait**. La nueva solución con estas modificaciones puede verse en el programa 5.2.

Programa 5.2 Monitor Productor/Consumidor simplificado

monitor *Productor_Consumidor*
var p, d : **int**
var *buf* : **array**$[0, N)$ **of** T
var c_{prod}, c_{cons} : **cond**

$$\theta \doteq p = 0 \wedge d = 0 \wedge N > 0$$

procedure Prod($x : T$)
do $\neg p - d < N \mapsto$
wait.c_{prod}
od;
$buf.(p \ mod \ N) := x$;
$p := p + 1$;
signal.c_{cons}
end

procedure Cons(**var** $y : T$)
do $\neg p - d > 0 \mapsto$
wait.c_{cons}
od;
$y := buf.(p \ mod \ N)$;
$d := d - 1$;
signal.c_{prod}
end

La eliminación de estas señalizaciones produce una solución más eficiente ya que reduce la cantidad de chequeos de guardas en los bucles que encierran las operaciones **wait**. Como veremos más adelante, nuestro método propuesto realiza estas mejoras de forma automática.

5.1.2 Señalamiento automático

Una alternativa al uso de variables de condición es especificar directamente las condiciones sobre el estado. Esta idea se corresponde con la implementación de protocolos de comunicación entre distintos procesos accediendo el monitor, como se menciona al final de la sección anterior. En este sentido los *monitores de señalamiento automático* fueron propuestos por Hoare [Hoa74] como una evolución de las regiones críticas condicionales. En los mismos se eliminan las variables de condición y la operación **signal** modificando la operación **wait** (la llamaremos **await** para diferenciarla de la anterior), la cual toma como parámetro una expresión booleana de las variables:

await.<expresión booleana>

Si la misma no es satisfecha por el estado donde se ejecuta la sentencia, el proceso activo es bloqueado permitiendo que otros procesos tomen el control del monitor (el monitor pasa del estado cerrado al abierto). Un proceso bloqueado en la sentencia continúa su hilo de ejecución cuando el monitor pasa al estado abierto y se satisface la condición. A esta condición la llamaremos *guarda de la operación* **await**.

Para la implementación de esta clase de monitor se asocia a cada expresión booleana dentro de un **await** una cola, agregando una más para procesos esperando invocar un procedimiento. Sean $\mathsf{B}_1, \cdots, \mathsf{B}_m$ las guardas de las operaciones **await** que aparecen en un monitor, $c_1, \cdots, c_m$ las colas de cada una y e la cola de procesos esperando activar un procedimiento. La operación **await** puede especificarse como:

$$\mathbf{await}.\mathsf{B}_i \doteq c_i.\mathbf{enque}(\mathit{self})\ ;\ \mathbf{schedule}$$

con la siguiente definición del scheduler:

$$\begin{array}{lll} \mathbf{schedule} \doteq & \underline{\mathbf{if}}\ |e| > 0 & \mapsto \mathit{nextActive} := e.\mathbf{deque} \\ & \Box\ |c_1| > 0 \wedge \mathsf{B}_1 & \mapsto \mathit{nextActive} := c_1.\mathbf{deque} \\ & \quad\vdots & \quad\vdots \\ & \Box\ |c_m| > 0 \wedge \mathsf{B}_m & \mapsto \mathit{nextActive} := c_m.\mathbf{deque} \\ & \underline{\mathbf{fi}}; & \\ & \mathbf{run}.\mathit{nextActive} & \end{array}$$

La salida de un proceso activo desde un procedimiento puede especificarse simplemente como la llamada al scheduler:

$$\mathbf{return} \doteq \mathbf{schedule}$$

La elección del próximo proceso activo hecha por el scheduler es no determinista. Al igual que en los monitores con señalamiento implícito, distintas clases de monitores son propuestos en la literatura dependiendo de la existencia de prioridades entre la cola de entrada y las de las condiciones [BFC95].

En nuestra trabajo utilizaremos una clase especial de monitores con señalamiento automático sin prioridades entre las colas, en el cual las guardas de las operaciones **await** solo pueden contener variables del monitor que no sean locales a los procedimientos. Esto excluye expresiones que contengan variables parámetros de los procedimientos del monitor. El tipo de monitores es denominado *No Priority Restricted Automatic Signal* (*NPRAS*) según la clasificación en [BFC95]. Los monitores de esta clase que cumplen con esta limitación son denominados *No Priority Restricted Automatic Signal* (*NPRAS*). Cabe aclarar que esta restricción no disminuye la capacidad expresiva de estas construcciones (como se analiza en [BFC95]) y además será importante al momento de modelarlas con sistemas de transiciones.

En estos monitores la labor del scheduler puede adosarse fácilmente a las operaciones **await** y **return** ya que el chequeo de las expresiones booleanas es

global a los procedimientos:

$$
\begin{array}{lll}
\mathbf{await}.\mathsf{B}_i & \doteq & \underline{\mathbf{if}}\ \mathsf{B}_i \ \mapsto \mathbf{skip} \\
 & & \Box\ \neg\mathsf{B}_i \mapsto \\
 & & \quad c_i.\mathbf{enque}(self); \\
 & & \quad \underline{\mathbf{if}}\ |e| > 0 \mapsto \mathbf{run}.(e.\mathbf{deque}) \\
 & & \quad \langle \Box j\ : 1 \leqslant j \leqslant m \wedge j \neq i\ : \\
 & & \quad\quad |c_j| > 0 \wedge \mathsf{B}_j \mapsto \mathbf{run}.(c_j.\mathbf{deque})\ \rangle \\
 & & \quad \underline{\mathbf{fi}} \\
 & & \underline{\mathbf{fi}}
\end{array}
\tag{5.5}
$$

$$
\begin{array}{lll}
\mathbf{return} & \doteq & \underline{\mathbf{if}}\ |e| > 0 \mapsto \mathbf{run}.(e.\mathbf{deque}) \\
 & & \langle \Box j\ : 1 \leqslant j \leqslant m\ : \\
 & & \quad |c_j| > 0 \wedge \mathsf{B}_j \mapsto \mathbf{run}.(c_j.\mathbf{deque})\ \rangle \\
 & & \underline{\mathbf{fi}}
\end{array}
$$

Las expresiones cuantificadas dentro de los **if** son cuantificaciones generalizadas sobre el operador $\Box$ de elección no determinista. Notar que con este cambio, si un proceso ejecuta una espera con su guarda verdadera entonces continúa su ejecución, a diferencia de la especificación anterior de la operación **await**.

Estudios sobre estos monitores muestran que son más simples de utilizar en forma correcta que los monitores con señalamiento explícito [BFC95, BH05]. En los monitores con señalamiento no bloqueante (i.e. *Wait and Notify*) la causa de errores de programación, mencionada en la sección anterior, es el desconocimiento sobre cuando el proceso señalizado continuará su ejecución. Por ejemplo, un error común es escribir un procedimiento donde se ejecuta un **signal** cuando cierto predicado se satisface, mientras que después de esta operación, en la misma porción de código se la falsifica. Con respecto a los monitores con señalamiento bloqueante (i.e. *Signal and Urgent Wait*) puede suceder que el proceso señalizado altere el estado del monitor que tenía al momento de ejecutarse el **signal**. Esto produce que la escritura del programa después de la aparición de esta operación, sea susceptible a cometer errores ya que no se conoce de forma local el cambio de estado producido por el proceso señalizado. Con los monitores de señalamiento automático estos errores no aparecen ya que se elimina la operación **signal**.

Ejemplo 5.2 (Productor/Consumidor con señalamiento automático) En el programa 5.3 (pág. 160) se muestra la solución al problema de productores y consumidores utilizando monitores con señalamiento automático.

Comparando este programa con la solución con señalamiento explícito en el programa 5.1, puede notarse que es mucho más claro y más simple que la anterior. La versión simplificada en el programa 5.1 es similar a esta última, pero para obtenerla es necesario probar su corrección, lo cual es una tarea compleja que involucra la búsqueda de un invariante del monitor. □

En [BFC95] se derivan las condiciones de prueba que aseguran la corrección de las distintas clases de monitores. Las mismas son expresadas como una extensión de la semántica axiomática estándar mediante 3-uplas de Hoare. Esta

Programa 5.3 Monitor Productor/Consumidor señalamiento automático

```
monitor Productor_Consumidor
  var p, d : int;
  var buf : array[0, N) of T;
              θ ≐ p = 0 ∧ d = 0 ∧ N > 0
  procedure Prod(x : T)          procedure Cons(var y : T)
    await.(p − d < N);             await.(p − d > 0);
    buf.(p mod N) := x;            y := buf.(p mod N);
    p := p + 1                     d := d − 1
  end                            end
```

extensión de la lógica de Hoare se realiza postulando la existencia dos[1] predicados:

- el predicado I es el *invariante del monitor*; el mismo se satisface siempre que el monitor cambie del estado cerrado a abierto o viceversa, o sea al comenzar un procedimiento, antes y después de la operación **await**, y antes de finalizar el procedimiento (en la primitiva **return**).
- El predicado E que debe satisfacerse cada vez que un proceso despierte de la cola e comenzando la ejecución de un procedimiento. Para señalar esta condición se agrega la primitiva **enter** al comienzo de todos los procedimientos del monitor.

Al momento de verificar un monitor estos predicados deben ser encontrados. En el caso de monitores *NPRAS* las condiciones para las primitivas y operaciones **enter**, **await** y **return** serán:

$$\begin{aligned} &\{\mathsf{true}\}\ \mathbf{enter}\ \{I \wedge E\} \\ &\{I \wedge E\}\ \mathbf{await}.\mathsf{B}_i\ \{I \wedge \mathsf{B}_i\} \\ &\{I \wedge E\}\ \mathbf{return}\ \{\mathsf{false}\}\ . \end{aligned}$$

En el articulo mencionado anteriormente, estas condiciones son derivadas de las especificaciones en (5.5) dadas anteriormente. Cabe agregar que en este resultado los predicados I y E aparecen en conjunción, por lo que el segundo puede ser incluido en el primero:

$$\begin{aligned} &\{\mathsf{true}\}\ \mathbf{enter}\ \{I\} \\ &\{I\}\ \mathbf{await}.\mathsf{B}_i\ \{I \wedge \mathsf{B}_i\} \\ &\{I\}\ \mathbf{return}\ \{\mathsf{false}\}\ . \end{aligned} \tag{5.6}$$

Aunque las condiciones de prueba para esta clase de monitores son muy simples, lamentablemente los lenguajes de programación rara vez los implementan[2]. Esto es debido a los problemas de eficiencia que produce la evaluación de las

[1] En el caso de monitores con señalamiento explícito se agrega un predicado extra para la operación **signal**.

[2] El lenguaje de programación Edison [Han81] los implementa.

guardas mostradas en las especificaciones de las operaciones **await** y **return** en (5.5). Como veremos más adelante, nuestro trabajo intenta eliminar estos chequeos innecesarios de manera automática.

5.2 Antecedentes

Los monitores fueron propuestos y desarrollados conjuntamente por Brinch Hansen [Han73b] y C.A.R. Hoare [Hoa74]. La descripción inicial de los mismos hecha en estos trabajos es en la forma de monitores con señalamiento explícito, aunque Hoare, basándose en las experiencias anteriores con regiones críticas condicionales, sugiere que la versión con señalamiento automático es más simple de utilizar correctamente. Más tarde Brinch Hansen implementa los monitores con señalamiento automático en el lenguaje Edison [Han81].

Un trabajo que ataca directamente el problema de la eficiencia de los monitores con señalamiento automático es [Sch76], el cual ya fue mencionado en el capítulo anterior (sección 4.3). Aunque el mismo comienza analizando el problema de eficiencia de regiones críticas condicionales, finaliza aplicando las técnicas desarrolladas sobre esas construcciones a monitores.

En [BFC95] Peter A. Buhr y Michael Frontier realizan una taxonomía exhaustiva de los monitores con señalamiento automático y explícito, sugiriendo nuevas clases de monitores. Allí también desarrollan las reglas de prueba de los distintos tipos de monitores para asegurar su corrección, con las cuales se analiza su facilidad de uso y se proponen técnicas de programación acordes a cada uno. En ese trabajo además se demuestra la equivalencia expresiva entre las distintas clases de monitores.

En [BH05], Peter A. Buhr y Ashif S. Harji analizan la posibilidad de implementación de monitores con señalamiento automático a partir de monitores con señalamiento explícito. Para ello construyen simulaciones sobre estas implementaciones en diferentes lenguajes de programación. Las mismas son hechas a partir de reemplazos puramente sintácticos sobre los monitores con señalamiento automáticos (a la manera de la mostrada en la sección siguiente) y sin ningún tipo de mejoras como las desarrolladas en nuestro trabajo.

5.3 Implementación de monitores con señalamiento automático

Como mencionamos al principio de este capítulo, nuestro trabajo consiste en mejorar, de manera automática, implementaciones de monitores con señalamiento automático utilizando monitores de señalamiento explícito. En el capítulo anterior vimos como la técnica SBD brindaba implementaciones de regiones críticas condicionales con semáforos binarios (aunque poco eficientes). Aquí, de manera análoga, comenzaremos a describir la implementación de monitores de señalamiento automático de tipo NPRAS (*No Priority Restricted Automatic Signal*) con monitores de señalamiento explícito NPNB (No priority Non-blocking). La decisión de utilizar esta clase de monitores es que son brindados comúnmente en los lenguajes de programación, como ya se mencionó en la sección 5.1.1. Implementaremos monitores con señalamiento automático NPRAS ya que poseen una implementación simple con monitores NPNB. Esto es debido a la característica

de falta de prioridad entre las colas que comparten ambos. Para implementar monitores con prioridad utilizando otros sin esta característica es necesario implementar colas extras de procesos, como se indica en [BH05].

La implementación que utilizaremos está basada en el estilo de codificación *signal as hint* que mencionamos en la sección 5.1.1 y puede encontrarse en la literatura [And91, BH05]. La idea general es asociar cada guarda de una operación **await** de un monitor con señalamiento automático, a una variable de condición en la implementación. Sean $\mathsf{B}_1, \cdots, \mathsf{B}_m$ las guardas de las operaciones **await** que aparecen en un monitor. Por cada una utilizaremos una variable de condición $c_1, \cdots, c_m$ respectivamente, las cuales deben ser declaradas dentro del monitor con señalamiento explícito en la implementación. Con este agregado, cada aparición de una operación **await**.B_i en el monitor a implementar será reemplazada de manera sintáctica por:

$$\begin{array}{rcl} \textbf{await}.\mathsf{B}_i & \doteq & \underline{\textbf{if}}\ \ \mathsf{B}_i \mapsto \textbf{skip} \\ & & \square\ \neg\mathsf{B}_i \mapsto \\ & & \quad \textbf{for}\ (j = 1, \cdots, m \wedge j \neq i) \\ & & \quad\quad \textbf{signal}.c_j; \\ & & \quad \textbf{wait}.c_i; \\ & & \quad \underline{\textbf{do}}\ \neg\mathsf{B}_i \mapsto \textbf{wait}.c_i\ \underline{\textbf{od}} \\ & & \underline{\textbf{fi}} \end{array} \tag{5.7}$$

Además, a la salida de cada procedimiento se agregará la señalización a todos los procesos esperando por el cumplimiento de su guarda:

$$\begin{array}{rcl} \textbf{return} & \doteq & \textbf{for}\ (j = 1, \cdots, m) \\ & & \quad \textbf{signal}.c_j; \\ & & \textbf{return} \end{array} \tag{5.8}$$

Notar que con estas manipulaciones sintácticas el monitor con señalamiento automático escrito en el programa 5.3 (solución del problema Productor/Consumidor) es implementado con el programa 5.1 obtenido utilizando el estilo de código *signal as hint*.

Como mostramos con el programa 5.2 la implementación obtenida con estas modificaciones sintácticas es poco eficiente, en el sentido que pueden eliminarse señalamientos superfluos. Para encontrarlos, primero modelaremos estas implementaciones con sistemas de transiciones.

5.4 Monitores como sistema de transiciones

En esta sección mostraremos como generar un sistema de transiciones a partir de la implementación con monitores de señalamiento explícito desarrollada en la sección anterior. Para ello procederemos a la manera que lo hicimos con las implementaciones SBD en el capítulo anterior. En aquel caso, las implementaciones con semáforos tenían un estructura regular cualquiera sea el problema a resolver, la cual se indicó en el programa 4.3 (pág. 111). Más específicamente, las operaciones de sincronización P.s y V.s aparecen solo dentro del protocolo de entrada (**if** inicial de la implementación SBD) o en el protocolo de salida (**if** final). Esta estructura en las implementaciones SBD permitió enumerar las transiciones posibles para formar el sistema de transiciones (sección 4.4, pág. 118).

En el caso de implementaciones de monitores con señalamiento automático la estructura de los programas ya no muestran esta regularidad: las operaciones de sincronización **await**.B a implementar pueden aparecer en cualquier lugar dentro de los procedimientos del monitor. Estas operaciones tienen la misma jerarquía gramatical que cualquier sentencia del programa, lo cual permite que el programador las introduzca por ejemplo dentro de las sentencias de una alternativa **if**. Por esta razón vamos a generar el sistema de transiciones de forma inductiva sobre la estructura gramatical de los procedimientos.

De todas maneras, la modelización con sistemas de transiciones seguirá el estilo de la que realizamos para las implementaciones SBD. En este sentido, conservaremos aquel concepto de *sección* como traza de ejecución atómica dentro del monitor, gracias a la exclusión mutua que estos brindan en la ejecución de sus procedimientos. Cada una estas secciones será modeladas con una transición, lo cual permitirá una granularidad mayor de los sistemas con su correspondiente simplificación en su tratamiento.

De igual forma, conservaremos la idea de asociar a cada locación el argumento de la operación bloqueante: en el capítulo anterior cada locación se identificó con un semáforo, mientras que en el capítulo actual asociaremos cada locación a una variable de condición. Mas específicamente, dado un monitor con señalamiento automático a implementar, donde dentro de sus procedimientos aparecen m operaciones **await** con predicados parámetro $\mathsf{B}_1, \cdots, \mathsf{B}_m$, comenzaremos definiendo las locaciones de la siguiente forma:

1. Implementaremos el monitor con uno de señalamiento explícito como se indicó en la sección anterior. El nuevo monitor tendrá m variables de condición $c_1, \cdots, c_m$ asociadas a los predicados parámetro anteriores.

2. Cada una de las variables de condición será identificada con una locación del sistema de transiciones que modelará el comportamiento de esta implementación. En particular, las transiciones que tengan estas locaciones de salida modelarán las secciones que comienzan su ejecución dentro de las implementaciones de las operaciones **await**. Por ejemplo, una transición podrá comenzar en una locación c_i si la condición B_i se cumple, modelando el comienzo de ejecución de una sección por el desbloqueo de una operación **await**.B_i.

3. Además, agregaremos una locación e para procesos intentando invocar algún procedimientos (a la manera de la locación s_m en la modelización SBD). Las transiciones que la tengan como locación de salida, modelarán las secciones al comienzo de la invocación de un procedimiento.

Al finalizar este procedimiento tendremos definido el conjunto de locaciones $\mathcal{L} \doteq \{e, c_1, \cdots, c_m\}$ del sistema de transiciones.

Como ya mencionamos, el sistema de transiciones será generado de forma inductiva sobre la estructura gramatical de los procedimientos. Esta derivación se hará sobre el monitor implementado, o sea desplegando de manera puramente sintáctica la implementación de las operaciones en la sección anterior sobre el monitor con señalamiento automático a implementar. Para simplificar la exposición utilizaremos un sistema de reglas al estilo *cálculo de secuentes* adaptado a nuestra tarea: un secuente será de la forma $S \vdash \tau$ con S un programa y τ una transición, indicando que a partir de un programa S se derivará la transición τ.

La misma tendrá como sentencia una asignación guardada, por lo que siempre tendrá la forma $(l,\ \square\ B \mapsto P_1; \cdots; P_k,\ l')$ con $P_1, \cdots, P_k$ sentencias determinista y totales.

Una regla tendrá la forma:

$$\frac{\text{premisa}_1 \cdots \text{premisa}_n}{\text{conclusión}}\ \textit{nombre} \qquad \text{condición lateral}$$

donde las premisas y la conclusión serán secuentes y la condición lateral un predicado. Esta indica que si se puede generar las transiciones en las premisas entonces se puede derivar la de la conclusión. A la manera usual, se utilizarán metavariables en la reglas para denotar un esquema de reglas. Las metavariables las escribiremos en mayúscula itálica. En el caso de que no haya premisas se obviará la linea horizontal a la manera de un axioma esquema.

De forma introductoria comenzaremos a definir la regla para la composición secuencial de dos programas $S_1; S_2$. Si de ambos programas por separado se pueden generar dos transiciones respectivamente, del programa compuesto se derivará un transición concatenando las anteriores y propagando hacia atrás la guarda con el transformador wlp (a la manera que lo hicimos en el capítulo anterior):

$$\frac{S_1 \vdash\ (L,\ \square\ B_1 \mapsto P_1,\ \bullet) \quad S_2 \vdash\ (\bullet,\ \square\ B_2 \mapsto P_2,\ L')}{S_1; S_2 \vdash\ (L,\ \square\ B_1 \wedge \text{wlp}.P_1.B_2 \mapsto P_1; P_2,\ L')}\ \textit{Seq1} \tag{5.9}$$

Notemos que utilizamos una *locación auxiliar* $\bullet$ la cual denota un nodo intermedio de la transición. Esto se hace para diferenciar las locaciones antes mencionadas $(e, c_1, \cdots, c_m)$, las cuales quedarán en el sistema de transiciones final. Como veremos, la locación auxiliar será eliminada por concatenación de transiciones en las diferentes reglas, lo cual nos permitirá aumentar la granularidad del sistema. Al final de esta sección presentaremos un ejemplo de derivación de transiciones desde la implementación de un monitor con señalamiento automático que mostrará este hecho.

Observemos que al realizar una composición secuencial $S_1; S_2$, dentro de cada programa en forma individual, pueden haberse generado transiciones que no se concatenan a la manera que lo hicimos en esta regla. Esto es debido a que dentro de los programas pueden aparecer operaciones **await** que generen transiciones con locaciones en la variables de condición $c_1, \cdots, c_m$ o en la locación de entrada e. Por ejemplo, una transición con locación de llegada c_i generada en el programa S_1 no debe concatenarse con una en el programa S_2 que tenga locación de salida c_j ya que perderíamos la información de las transiciones generadas por las operaciones bloqueantes, cuyas reglas veremos más adelante. Es por ello necesario agregar las siguientes reglas para la composición secuencial:

$$\frac{S_1 \vdash\ (L,\ \square\ B \mapsto P,\ L')}{S_1; S_2 \vdash\ (L,\ \square\ B \mapsto P,\ L')}\ \textit{Seq2} \quad \text{si } L' \neq \bullet$$

Esta regla significa que en el caso que desde S_1 se haya generado una transición con locación de llegada distinta de la locación auxiliar $\bullet$ (o sea alguna en $\{e, c_1, \cdots, c_m\}$), entonces la transición se preserva. La locación de salida puede ser cualquiera, incluso $\bullet$. El caso simétrico ocurre con S_2, donde se requerirá como condición lateral que la locación de salida de de la transición generada

por S_2 sea distinta a la locación auxiliar:

$$\frac{S_2 \vdash\ (L,\ \square\, B \mapsto P,\ L')}{S_1; S_2 \vdash\ (L,\ \square\, B \mapsto P,\ L')}\ \textit{Seq3} \quad \text{si } L \neq \bullet$$

Uno de los casos base en la gramática estructural de los programas será la asignación, la cual derivará su transición asociada por el siguiente axioma esquema:

$$v := H \vdash\ (\bullet,\ \square\, \mathsf{true} \mapsto v := H,\ \bullet) \quad \textit{Assign}$$

En este caso, las locaciones de entrada y salida de la transición generada serán la locación auxiliar, las cual será eliminada por la primera regla *Seq1*.

Para las apariciones de sentencias **skip**, el axioma de generación será simplemente:

$$\mathbf{skip} \vdash\ (\bullet,\ \square\, \mathsf{true} \mapsto \mathbf{skip},\ \bullet) \quad \textit{Skip}$$

la cual produce una transición con la sentencia guardada equivalente.

Una sentencia que será utilizada como parte de los procedimientos de los monitores será la suposición (ejemplo 1.1, pág. 9). La misma servirá para indicar precondiciones en los procedimientos, como veremos en los casos de prueba presentados la sección 5.8. La regla generará una transición con la suposición como guarda de la sentencia **skip**:

$$[B] \vdash\ (\bullet,\ \square\, B \mapsto \mathbf{skip},\ \bullet) \quad \textit{Assum}$$

Al comienzo de un procedimiento se generarán transiciones con locación de salida e ya que esta cola contiene los procesos que intentan invocarlo. Para formalizar la derivación de estas transiciones agregaremos la primitiva **enter**, la cual debe aparecer al comienzo de todos los procedimientos. El axioma sobre esta primitiva será simplemente

$$\mathbf{enter} \vdash\ (e,\ \square\, \mathsf{true} \mapsto \mathbf{skip},\ \bullet) \quad \textit{Enter}$$

Notar que eventualmente esta transición será premisa de la regla *Seq1* junto con las generadas por el resto del procedimiento. Esto elimina las locaciones de entrada auxiliares. Más adelante veremos que las reglas para **return** posibilitan también la eliminación de las locaciones de salida auxiliares a la finalización de los procedimiento.

Para la sentencia de bifurcación condicional **<u>if</u>** tendremos la siguiente regla:

$$\frac{S_i \vdash\ (\bullet,\ \square\, B \mapsto P,\ L')}{\underline{\mathbf{if}}\ \langle \square j\ : 1 \leqslant j \leqslant n :\ C_j \mapsto S_j \rangle\ \underline{\mathbf{fi}}\ \vdash\ (\bullet,\ \square\, C_i \wedge B \mapsto P,\ L')}\ \textit{If1} \qquad (5.10)$$

la cual genera una transición desde S_i si su guarda se verifica. Hay que tener en cuenta que esta regla se puede aplicar solamente si la locación de salida en la transición de la premisa es la locación auxiliar. Esta restricción posibilita la eventual concatenación con transiciones previamente generadas en el cuerpo del programa. Por otra parte, al igual que con las reglas *Seq2* y *Seq3*, declararemos una regla extra para el caso de transiciones con locación de salida

en $\{e, c_1, \cdots, c_m\}$ generadas en S_i desde las operaciones bloqueantes. En este caso las transiciones también serán preservadas por las reglas:

$$\frac{S_i \vdash\ (L,\ \square\ B \mapsto P,\ L')}{\underline{\mathbf{if}}\ \langle \square j\ : 1 \leqslant j \leqslant n :\ C_j \mapsto S_j \rangle\ \underline{\mathbf{fi}}\ \vdash\ (L,\ \square\ B \mapsto P,\ L')}\ \mathit{If2} \qquad \text{si } L \neq \bullet$$

Hasta ahora las reglas presentadas corresponden a sentencias y primitivas que no alteran las colas de espera del monitor. Estas serán modeladas en nuestro sistema de transiciones. De acuerdo a la especificación de monitores con señalamiento explícito (sección 5.1.1) se utilizan las colas $w_1, \cdots, w_m$ de procesos esperando en operaciones **wait** y las colas de procesos señalizados $s_1, \cdots, s_m$, ambas asociadas a las variables de condición $c_1, \cdots, c_m$. Para incluirlas en los sistemas de transiciones modelaremos solo su longitud a manera de abstracción. De esta forma las colas serán representadas por enteros positivos que serán incrementados o decrementados según se apliquen las operaciones **enque** o **deque** respectivamente. Notar que esta abstracción ya fue utilizada en el capítulo anterior con los contadores b_i, los cuales indicaban la cantidad de procesos esperando en las operaciones bloqueantes P sobre cada semáforo s_i. En aquel caso, estas variables estaban incluidas como parte de las implementaciones SBD. En el caso de los monitores las agregaremos de manera explícita para poder modelar su comportamiento.

Tomemos la implementación de la operación **await** en (5.7) desplegando la especificación de **signal** en (5.1) (pág. 154) y reemplazando las operaciones sobre las colas por decrementos e incrementos en sus tamaños (se incrementan las colas de señalizado y se decrementan las de procesos en espera). Además separaremos la parte de la implementación correspondiente a la codificación *signal as hint* (final donde se espera y se ejecuta el bucle) introduciéndola en el programa separado $\mathbf{waitLoop}_i$:

$$\begin{array}{rcl} \mathbf{await}.\mathsf{B}_i & \doteq & \underline{\mathbf{if}}\ \ \mathsf{B}_i \mapsto \mathbf{skip} \\ & & \square\ \neg\mathsf{B}_i \mapsto \\ & & \quad \mathbf{for}\ (j = 1, \cdots, m \wedge j \neq i) \\ & & \quad\quad \underline{\mathbf{if}}\ w_j > 0 \mapsto w_j, s_j := w_j - 1, s_j + 1 \\ & & \quad\quad \square\ \ w_j = 0 \mapsto \mathbf{skip} \\ & & \quad\quad \underline{\mathbf{fi}}; \\ & & \quad \mathbf{waitLoop}_i \\ & & \underline{\mathbf{fi}} \end{array} \qquad (5.11)$$

con

$$\begin{array}{rcl} \mathbf{waitLoop}_i & \doteq & \mathbf{wait}.c_i; \\ & & \underline{\mathbf{do}}\ \neg\mathsf{B}_i \mapsto \mathbf{wait}.c_i\ \underline{\mathbf{od}} \end{array}$$

Notemos que el bucle **for** es acotado por lo que puede ser desplegado obteniéndose la composición secuencial de $m-1$ sentencias **if** (que aparecen dentro de este bucle). Por lo tanto, de manera esquemática el bucle puede ser escrito

como el siguiente programa equivalente:

$$
\begin{array}{rl}
 & \underline{\textbf{if}}\ w_1 > 0 \mapsto w_1, s_1 := w_1 - 1, s_1 + 1 \\
 & \Box\ w_1 = 0 \mapsto \textbf{skip} \\
 & \underline{\textbf{fi}}; \\
 & \vdots \\
 & \underline{\textbf{if}}\ w_j > 0 \mapsto w_j, s_j := w_j - 1, s_j + 1 \\
(j \neq i) & \Box\ w_j = 0 \mapsto \textbf{skip} \\
 & \underline{\textbf{fi}}; \\
 & \vdots \\
 & \underline{\textbf{if}}\ w_m > 0 \mapsto w_m, s_m := w_m - 1, s_m + 1 \\
 & \Box\ w_m = 0 \mapsto \textbf{skip} \\
 & \underline{\textbf{fi}}
\end{array}
$$

o utilizando la notación de cuantificación generalizada sobre el operador de composición secuencial:

$$
\begin{array}{l}
\langle (;)\, j\ : 1 \leqslant j \leqslant m \wedge j \neq i\ : \\
\quad \underline{\textbf{if}}\ w_j > 0 \mapsto w_j, s_j := w_j - 1, s_j + 1 \,\Box\, w_j = 0 \mapsto \textbf{skip}\ \underline{\textbf{fi}} \rangle
\end{array}
$$

De esta manera, hasta la aparición de $\textbf{waitLoop}_i$, el programa puede ser tratado por las reglas anteriormente definidas.

Desplegando la definición del scheduler en (5.3) sobre la especificación de la operación **wait** en (5.2) (pág. 155) tenemos:

$$
\begin{array}{rcl}
\textbf{wait}.c_i & \doteq & w_i.\textbf{enque}(\mathit{self}); \\
 & & \underline{\textbf{if}}\ |e| > 0 \ \ \mapsto \mathit{nextActive} := e.\textbf{deque} \\
 & & \Box\ \ |s_1| > 0 \ \mapsto \mathit{nextActive} := s_1.\textbf{deque} \\
 & & \quad\quad \vdots \quad\quad\quad\quad\quad\quad \vdots \\
 & & \Box\ \ |s_m| > 0 \mapsto \mathit{nextActive} := s_m.\textbf{deque} \\
 & & \underline{\textbf{fi}}; \\
 & & \textbf{run}.\mathit{nextActive}
\end{array}
$$

En este caso aplicaremos dos tipos de abstracciones a las colas: la ya mencionada, donde las colas $s_1, \cdots, s_m$ y $w_1, \cdots, w_m$ son modeladas por su longitud y abstraeremos la cola e eliminándola de la representación. Esta última abstracción no modificará el comportamiento operacional del sistema de transiciones ya que siempre podrá ingresar un proceso intentando invocar un procedimiento de manera no determinista. Notar que en las implementaciones SBD tampoco se utilizan contadores para procesos bloqueados en el semáforo de entrada s_m. Además, señalaremos la ejecución del próximo proceso como $\textbf{run}.c$, con c la variable de condición donde el proceso espera o la cola de entrada. Con estas

modificaciones el anterior programa queda:

$$
\begin{array}{rcl}
\mathbf{wait}.c_i & \doteq & w_i := w_i + 1; \\
 & & \underline{\mathbf{if}}\ \mathsf{true} \quad \mapsto \mathbf{run}.e \\
 & & \Box\ s_1 > 0 \ \mapsto s_1 := s_1 - 1; \mathbf{run}.c_1 \\
 & & \quad \vdots \qquad\qquad \vdots \\
 & & \Box\ s_m > 0 \mapsto; s_m := s_{c_m} - 1; \mathbf{run}.c_m \\
 & & \underline{\mathbf{fi}}
\end{array}
$$

Desplegando este programa solo en la primer operación **wait** dentro de $\mathbf{waitLoop}_i$ tenemos:

$$
\begin{array}{rcl}
\mathbf{waitLoop}_i & \doteq & w_i := w_i + 1; \\
 & & \underline{\mathbf{if}}\ \mathsf{true} \quad \mapsto \mathbf{run}.e \\
 & & \Box\ s_1 > 0 \ \mapsto s_1 := s_1 - 1; \mathbf{run}.c_1 \\
 & & \quad \vdots \qquad\qquad \vdots \\
 & & \Box\ s_m > 0 \mapsto; s_m := s_{c_m} - 1; \mathbf{run}.c_m \\
 & & \underline{\mathbf{fi}} \\
 & & \underline{\mathbf{do}} \neg \mathsf{B}_i \mapsto \mathbf{wait}.c_i\ \underline{\mathbf{od}}
\end{array}
$$

Las operaciones $\mathbf{run}.c$ denotan el cambio de ejecución a otro proceso esperando en la cola c. Dentro del sistema de transiciones, este cambio de ejecución de proceso será modelado con una transición saliente de la locación c (recordar que las locaciones se identifican con las colas). Por lo tanto, del anterior programa puede extraerse la siguiente regla para el caso que despierte un proceso intentando ejecutar un procedimiento por la activación de la primer guarda del **if**:

$$
\mathbf{waitLoop}_i \vdash\ (\bullet,\ \Box\, \mathsf{true} \mapsto w_i := w_i + 1,\ e) \quad \mathit{WakeEnter} \tag{5.12}
$$

En el caso que se despierte un proceso por ejecución de $\mathbf{run}.c_j$ las reglas serán:

$$
\mathbf{waitLoop}_i \vdash\ (\bullet,\ \Box\, s_j > 0 \mapsto w_i, s_j := w_i + 1, s_j - 1,\ c_j) \quad \mathit{WakeWait} \tag{5.13}
$$

con $j = 1, \cdots, m$. Notar que se a propagado la guarda $s_j > 0$ para conformar la sentencia de asignación guardada en la transición.

Las anteriores son las únicas reglas generadas por los programas $\mathbf{waitLoop}_i$ antes de bucle **do**. Desde la aparición de este bucle el proceso despertará (por medio de alguna señalización) intentando verificar su guarda B_i. Para analizar las transiciones generadas desde este punto, desplegaremos solo la operación

wait dentro del bucle:

$$
\begin{array}{rcl}
\textbf{waitLoop}_i & \doteq & \textbf{wait}.c_i; \\
& & \underline{\textbf{do}}\ \neg \mathsf{B}_i\ \mapsto \\
& & \quad w_i := w_i + 1; \\
& & \quad \underline{\textbf{if}}\ \ \mathsf{true} \quad\ \mapsto \textbf{run}.e \\
& & \quad \square\ \ s_1 > 0\ \ \mapsto s_1 := s_1 - 1; \textbf{run}.c_1 \\
& & \qquad\quad \vdots \qquad\qquad\qquad \vdots \\
& & \quad \square\ \ s_m > 0 \mapsto; s_m := s_{c_m} - 1; \textbf{run}.c_m \\
& & \quad \underline{\textbf{fi}} \\
& & \underline{\textbf{od}}
\end{array}
$$

En el caso que la condición B_i sea satisfecha el proceso esperando en el **wait** inicial despierta. Para modelar este comportamiento daremos la siguiente regla:

$$\textbf{waitLoop}_i \vdash\ (c_i,\ \square\ \mathsf{B}_i \mapsto \textbf{skip},\ \bullet) \quad \textit{Wake}$$

En el caso que aquella condición no se satisfaga, el proceso ejecutará el cuerpo del bucle iniciando la ejecución de otro esperando por invocar un procedimiento (cola e) o esperando en una operación **wait**. El primer caso modelará la ejecución de la primer guarda de la sentencia **if**:

$$\textbf{waitLoop}_i \vdash\ (c_i,\ \square\ \neg\mathsf{B}_i \mapsto w_i := w_i + 1,\ e) \quad \textit{LoopEnter}$$

y para las otras guardas las reglas serán:

$$\textbf{waitLoop}_i \vdash\ (c_i,\ \square\ \neg\mathsf{B}_i \wedge s_j > 0 \mapsto w_i, s_j := w_i + 1, s_j - 1,\ c_j) \quad \textit{LoopWait}$$

con $j = 1, \cdots, m$. Aquí también propagamos la condición $s_j > 0$ para formar la guarda de la asignación guardada dentro de la transición.

Por último veremos las reglas correspondientes a la terminación en la ejecución de un procedimiento señalado con la primitiva **return**. Tomando la implementación de la misma en (5.8) (pág. 162), desplegando la especificación de **signal** en (5.1) (pág. 154) y haciendo la abstracción de colas como enteros positivos, tenemos el siguiente programa:

$$
\begin{array}{rcl}
\textbf{return} & \doteq & \textbf{for}\ (j = 1, \cdots, m) \\
& & \quad \underline{\textbf{if}}\ w_j > 0 \mapsto w_j, s_j := w_j - 1, s_j + 1 \\
& & \quad \square\ \ w_j = 0 \mapsto \textbf{skip} \\
& & \quad \underline{\textbf{fi}}; \\
& & \textbf{return}
\end{array}
\tag{5.14}
$$

Al igual que con la implementación de la operación **await** el bucle **for** puede ser expandido en una composición secuencial de m sentencias condicionales, por lo tanto esta parte del programa puede ser tratado por el conjunto de reglas ya definidas. Queda por ver la primitiva **return** para monitores de señalamiento explícito en el lado derecho de la definición. Para ella desarrollaremos nuevas reglas desplegando su especificación en (5.4) (pág. 155) y realizando la abstracción

sobre las colas:

$$\begin{array}{rcll} \textbf{return} & \doteq & \underline{\textbf{if}}\ \mathsf{true} & \mapsto \textbf{run}.e \\ & & \square\ s_1 > 0 & \mapsto s_1 := s_1 - 1; \textbf{run}.c_1 \\ & & \quad \vdots & \quad \vdots \\ & & \square\ s_m > 0 & \mapsto s_m := s_{c_m} - 1; \textbf{run}.c_m \\ & & \underline{\textbf{fi}} & \end{array}$$

Las reglas son inducidas por este **if** final y son similares a las previamente definidas para el inicio de $\textbf{waitLoop}_i$. Para la primer guarda, donde se libera un proceso nuevo, la regla será:

$$\textbf{return} \vdash\ (\bullet,\ \textbf{skip},\ e) \quad \textit{ReturnEnter}$$

y para las guardas siguientes (se libera un proceso esperando en una operación **wait**) tendremos:

$$\textbf{return} \vdash\ (\bullet,\ \square\ s_j > 0 \mapsto s_j := s_j - 1,\ c_j) \quad \textit{ReturnWait}$$

En el apéndice 5.A (pág. 203) se hace un listado de todas las reglas desarrolladas.

Las reglas de generación de transiciones serán aplicadas a todos lo procedimientos del monitor por separado para obtener el conjunto de transiciones $\mathcal{T}$ del sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ que lo modela. Si tenemos la implementación de un monitor con señalamiento automático de la forma:

$$\begin{array}{l} \underline{\textbf{monitor}}\ \textit{Nombre_monitor} \\ \quad \begin{array}{lcl} \underline{\textbf{procedure}}\ op_1() & & \underline{\textbf{procedure}}\ op_n() \\ \quad S_1 & \cdots & \quad S_n \\ \underline{\textbf{end}} & & \underline{\textbf{end}} \end{array} \end{array}$$

donde las apariciones de las operaciones **await** son reemplazadas por la implementación en (5.11), entonces el conjunto $\mathcal{T}$ estará formado por las transiciones que se deriven de los cuerpos S_i de los procedimientos:

$$\mathcal{T} \ \doteq\ \{\tau \mid \langle \exists\, i\ : 1 \leqslant i \leqslant n :\ S_i \vdash \tau \rangle\}$$

con el conjunto de sentencias $\mathcal{S}$ extraído de estas transiciones.

Notemos que los procedimientos se modelan sin ningún tipo de variables locales ni parámetros. Esta es una restricción impuesta en nuestro trabajo para simplificar el tratamiento del problema. En el caso que los procedimientos tengan este tipo de variables, las mismas no pueden ser modeladas estrictamente por nuestros sistemas de transiciones ya que la abstracción de las colas de procesos como contadores hace que aquellos ejecutando un mismo procedimiento sean indistinguibles entre sí (no se modela su *pila*). Este tratamiento produce sistemas de transiciones de tamaño tratable y suficientemente expresivos para poder ser utilizado sobre una gran cantidad de problemas. Sin embargo, problemas que impliquen el uso de variables locales, como por ejemplo *Disk schedulling* [And91, pág. 295] no pueden ser abordados con este modelado. El único camino para tratar este problema sin pérdida de información es incluir la pila

de variables locales de cada proceso en el sistema de transiciones. Necesariamente esto rompe con la abstracción de las colas de procesos antes mencionada. De todas maneras nuestro trabajo puede ser extendido incorporando sentencias **havoc** [DS90] sobre las variables locales como se indica en [BL05]. Cabe aclarar además, que los monitores NPRAS a implementar (ver sección 5.1.2) no admiten variables locales en las condiciones, pero las mismas pueden depender de este tipo de variables. Esto asegura su equivalencia expresiva con los monitores en general [BFC95].

A continuación veremos como se derivan las transiciones para el monitor que resuelve el problema clásico de productores y consumidores.

Ejemplo 5.3 (Transiciones del monitor Productor/Consumidor)
En el ejemplo 5.2 (pág. 159) vimos la solución de este problema con un monitor de señalamiento automático. Eliminando el acceso al buffer y realizando el mismo cambio de coordenadas que en el ejemplo 4.1 (pág. 111) obtenemos el monitor en el programa 5.4.

Programa 5.4 Monitor Productor/Consumidor señalamiento automático

```
monitor Productor_Consumidor
  var n : int;
                      θ ≐ n = 0 ∧ N > 0
  procedure Prod()                    procedure Cons()
    await.(n < N);                      await.(n > 0);
    n:=n+1                              n:=n−1
  end                                 end
```

Antes de comenzar a generar las transiciones implementaremos el monitor con uno de señalamiento explícito. Nos concentraremos en las transiciones derivadas a partir del procedimiento productor, el cual tiene la implementación dada en el programa 5.5 (pág. 172) donde señalamos las implementaciones de la operación **await**.$(n < N)$ y la primitiva **return** del procedimiento productor en el programa 5.4. Además hemos agregado la primitiva **enter** y representado con $\mathbf{waitLoop}_1$ el programa:

$$\mathbf{waitLoop}_1 \doteq \mathbf{wait}.c_1; \\ \underline{\mathbf{do}}\ \neg n < N \mapsto \mathbf{wait}.c_1\ \underline{\mathbf{od}}$$

Notar que desplegamos los bucles **for** donde se realizan las señalizaciones, tanto en la implementación de la operación **await** como en la primitiva **return** al final del procedimiento.

Comencemos entonces a derivar todas las transiciones posible a partir de las reglas desarrolladas en esta sección. Al comienzo del programa anterior aparece la primitiva **enter**, la cual, por aplicación de la regla *Enter* (pág. 165) se deriva la transición:

$$(e,\ \Box\ \mathsf{true} \mapsto \mathbf{skip},\ \bullet)\ , \qquad (5.15)$$

donde la locación auxiliar $\bullet$ será eliminada más adelante.

La sentencia siguiente es el **if** externo en la implementación del **await**. En su primera rama con condición $n < N$ aparece la sentencia **skip**. Desde la misma

Programa 5.5 Implementación del procedimiento productor

procedure Prod()
enter;
if $n < N \mapsto$ **skip**
$\square\ \neg n < N \mapsto$
if $w_2 > 0 \mapsto w_2, s_2 := w_2 - 1, s_2 + 1$
$\square\ w_2 = 0 \mapsto$ **skip**
fi;
waitLoop$_1$
fi;

await.$(n < N)$

$n := n + 1$;
if $w_1 > 0 \mapsto w_1, s_1 := w_1 - 1, s_1 + 1$
$\square\ w_1 = 0 \mapsto$ **skip**
fi;
if $w_2 > 0 \mapsto w_2, s_2 := w_2 - 1, s_2 + 1$
$\square\ w_2 = 0 \mapsto$ **skip**
fi;
return

return

end

se puede derivar la siguiente transición por aplicación de la regla *Skip*:

$$(\bullet,\ \square\ \mathsf{true} \mapsto \textbf{skip},\ \bullet)\ ,$$

mediante la cual, por aplicación de la regla *If1* y teniéndola como premisa, se deriva la siguiente transición:

$$(\bullet,\ \square\ n < N \mapsto \textbf{skip},\ \bullet)\ ,$$

donde hemos simplificado la guarda $n < N \wedge \mathsf{true}$, resultado de la regla, por $n < N$.

Esta última transición puede ser concatenada con la primera (generada por la regla *Enter*) aplicando la regla *Seq1* (pág. 164), obteniéndose la siguiente:

$$(e,\ \square\ \mathsf{true} \wedge n < N \mapsto \textbf{skip}; \textbf{skip},\ \bullet)\ .$$

Simplificando la guardas y la sentencia **skip** redundante obtenemos:

$$(e,\ \square\ n < N \mapsto \textbf{skip},\ \bullet)\ . \tag{5.16}$$

Esta transición sera concatenada con aquellas que se deriven desde la sentencia de asignación $n := n + 1$ después de la implementación del **await**.

A continuación veremos las transiciones generadas dentro de la segunda rama de esta sentencia **if** (con condición $\neg n < N$). En la misma aparece otra sentencia **if** interna donde se produce la señalización (representada por las modificaciones en las colas). Por aplicación de las reglas *Assign* y *Skip* (en las ramas de este último **if**) se obtienen dos transiciones que serán premisas de la

regla *If1*, obteniéndose las siguientes que modelan este **if** interno:

$$(\bullet,\ \square\, w_2 > 0 \mapsto w_2, s_2 := w_2 - 1, s_2 + 1,\ \bullet)\ ,$$
$$(\bullet,\ \square\, w_2 = 0 \mapsto \textbf{skip},\ \bullet)\ .$$

Después de esta sentencia condicional aparece el programa $\mathbf{waitLoop}_1$. Aplicando las reglas *WakeEnter* y *WakeWait* (pág. 168) obtenemos las tres transiciones:

$$(\bullet,\ \square\, \mathsf{true} \mapsto w_1 := w_1 + 1,\ e)\ ,$$
$$(\bullet,\ \square\, s_1 > 0 \mapsto w_1, s_1 := w_1 + 1, s_1 - 1,\ c_1)\ ,$$
$$(\bullet,\ \square\, s_2 > 0 \mapsto w_1, s_2 := w_1 + 1, s_2 - 1,\ c_2)\ .$$

Cada una de ellas se pueden concatenar con las anteriores (derivadas desde el **if** interno) utilizándolas como premisas de la regla *Seq1*. Se obtienen así seis nuevas transiciones:

$$(\bullet,\ \square\, w_2 > 0 \mapsto w_2, s_2 := w_2 - 1, s_2 + 1; w_1 := w_1 + 1,\ e)\ ,$$
$$(\bullet,\ \square\, w_2 > 0 \wedge s_1 > 0 \mapsto w_2, s_2 := w_2 - 1, s_2 + 1; w_1, s_1 := w_1 + 1, s_1 - 1,\ c_1)\ ,$$
$$(\bullet,\ \square\, w_2 > 0 \wedge s_2 > -1 \mapsto w_2, s_2 := w_2 - 1, s_2 + 1; w_1, s_2 := w_1 + 1, s_2 - 1,\ c_2)\ ,$$
$$(\bullet,\ \square\, w_2 = 0 \mapsto w_1 := w_1 + 1,\ e)\ ,$$
$$(\bullet,\ \square\, w_2 = 0 \wedge s_1 > 0 \mapsto w_1, s_1 := w_1 + 1, s_1 - 1,\ c_1)\ ,$$
$$(\bullet,\ \square\, w_2 = 0 \wedge s_2 > 0 \mapsto w_1, s_2 := w_1 + 1, s_2 - 1,\ c_2)\ .$$

Las primeras tres transiciones corresponden a las secciones donde se señaliza a un proceso consumidor y las restantes donde no hay procesos de este tipo esperando (por lo que las señalizaciones se pierden). Notar que se ha realizado la propagación, mediante el transformador wlp, de las guardas $s_1 > 0$ y $s_2 > 0$ en las transiciones generadas (segunda, tercera, quinta y sexta transición). Además hemos simplificado las guardas de las transiciones resultado como lo hicimos anteriormente.

Notemos también que en la guarda de la tercera transición aparece el termino conjuntivo $s_2 > -1$. El mismo puede ser eliminado ya que la longitud de las colas es siempre positivo para cualquier estado. Además, en las asignaciones del comando guardado de esta transición, la variable s_2 es incrementada y decrementada por lo que puede ser eliminada. Realizando estas simplificaciones y convirtiendo las asignaciones secuenciales en múltiples podemos escribir las transiciones anteriores de forma más compacta:

$$(\bullet,\ \square\, w_2 > 0 \mapsto w_2, s_2, w_1 := w_2 - 1, s_2 + 1, w_1 + 1,\ e)\ ,$$
$$(\bullet,\ \square\, w_2 > 0 \wedge s_1 > 0 \mapsto w_2, s_2, w_1, s_1 := w_2 - 1, s_2 + 1, w_1 + 1, s_1 - 1,\ c_1)\ ,$$
$$(\bullet,\ \square\, w_2 > 0 \mapsto w_2, w_1 := w_2 - 1, s_2 - 1,\ c_2)\ ,$$
$$(\bullet,\ \square\, w_2 = 0 \mapsto w_1 := w_1 + 1,\ e)\ ,$$
$$(\bullet,\ \square\, w_2 = 0 \wedge s_1 > 0 \mapsto w_1, s_1 := w_1 + 1, s_1 - 1,\ c_1)\ ,$$
$$(\bullet,\ \square\, w_2 = 0 \wedge s_2 > 0 \mapsto w_1, s_2 := w_1 + 1, s_2 - 1,\ c_2)\ .$$

Como veremos más adelante (sección 5.6), estas simplificaciones son importantes ya que reducen el tiempo de cálculo del método propuesto en este trabajo.

Cada una de estas últimas transiciones son generadas dentro de la segunda rama del **if** externo que implementa la operación **await**.$(n < N)$. Aplicando

la regla *If1* y concatenando los resultados con la transición generada por la primitiva **enter** en (5.15) obtenemos las siguientes transiciones finales:

$$
\begin{aligned}
&(e,\ \Box\, \neg n < N \wedge w_2 > 0 \mapsto w_2, s_2, w_1 := w_2 - 1, s_2 + 1, w_1 + 1,\ e)\ ,\\
&(e,\ \Box\, \neg n < N \wedge w_2 > 0 \wedge s_1 > 0 \mapsto\\
&\qquad w_2, s_2, w_1, s_1 := w_2 - 1, s_2 + 1, w_1 + 1, s_1 - 1,\ c_1)\ ,\\
&(e,\ \Box\, \neg n < N \wedge w_2 > 0 \mapsto w_2, w_1 := w_2 - 1, s_2 - 1,\ c_2)\ ,\\
&(e,\ \Box\, \neg n < N \wedge w_2 = 0 \mapsto w_1 := w_1 + 1,\ e)\ ,\\
&(e,\ \Box\, \neg n < N \wedge w_2 = 0 \wedge s_1 > 0 \mapsto w_1, s_1 := w_1 + 1, s_1 - 1,\ c_1)\ ,\\
&(e,\ \Box\, \neg n < N \wedge w_2 = 0 \wedge s_2 > 0 \mapsto w_1, s_2 := w_1 + 1, s_2 - 1,\ c_2)\ .
\end{aligned}
\tag{5.17}
$$

Estas transiciones aparecerán en el conjunto de transiciones $\mathcal{T}$ del sistema $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ que modela el monitor. Esto es debido a que son preservadas por las composiciones secuenciales subsiguientes por la aplicación exclusiva de las reglas *Seq2* y *Seq3* (son las únicas reglas que admiten como premisas transiciones con locaciones de entrada y salida distintas de la auxiliar). Las mismas serán ejecutadas por procesos que invocaron el procedimiento productor pero quedaron bloqueados por no cumplirse la condición $n < N$. En estos caso las transiciones finalizan en locaciones que modelan la activación de procesos esperando en las variables de condición c_1, c_2 (productor o consumidor respectivamente) o intentando invocar algún procedimiento desde la locación e.

Dentro de esta rama del **if** inicial se generan otras transiciones con locación de salida c_1 denotando procesos productores que, esperando en esta variable de condición, fueron señalizados y recomenzaron su ejecución. Dichas transiciones son generadas por las reglas *Wake*, *LoopEnter* y *LoopWait* (pág 169):

$$
\begin{aligned}
&(c_1,\ \Box\, n < N \mapsto \mathbf{skip},\ \bullet)\ ,\\
&(c_1,\ \Box\, \neg n < N \mapsto w_1 := w_1 + 1,\ e)\ ,\\
&(c_1,\ \Box\, \neg n < N \wedge s_1 > 0 \mapsto w_1, s_1 := w_1 + 1, s_1 - 1,\ c_1)\ ,\\
&(c_1,\ \Box\, \neg n < N \wedge s_2 > 0 \mapsto w_1, s_2 := w_1 + 1, s_2 - 1,\ c_2)\ .
\end{aligned}
\tag{5.18}
$$

Por aplicación de la regla *If2* (es la única regla para el **if** que admite locaciones de salida distintas de la auxiliar) todas estas transiciones serán preservadas. La primera será concatenada con las generadas por el resto del programa (desde la asignación $n := n + 1$ en adelante) ya que posee como locación de llegada la auxiliar. Las tres últimas formarán parte del conjunto de transiciones final $\mathcal{T}$ ya que serán preservadas por la composición secuencial (gracias a las reglas *Seq2* y *Seq3*). Las mismas denotan procesos productores que, esperando en la variable de condición c_1, recomenzaron su ejecución sin éxito ya que la condición $n < N$ no se satisfizo. Estas ejecuciones derivan del bucle **do** en la implementación de la operación **await** (pág. 162) resultado de la aplicación de la estrategia *signal as hint*.

Como ya mencionamos, la primer transición anterior y la transición en (5.16) (pág. 172) (derivada desde la sección que contiene la rama **skip** del **if** inicial) serán concatenadas con las generadas por el resto del programa. La próxima sentencia es la asignación $n := n + 1$, mediante la cual, aplicando la regla *Assign* y

concatenándola con aquellas por la regla *Seq1*, se obtienen las siguientes transiciones:

$$\begin{array}{l} (e,\ \square\, n < N \mapsto n{:=}n+1,\ \bullet)\ , \\ (c_1,\ \square\, n < N \mapsto n{:=}n+1,\ \bullet)\ . \end{array} \tag{5.19}$$

Las mismas serán concatenadas, mediante la última regla, con las generadas en la parte de implementación del **return** final. En esta parte del procedimiento se generan 12 transiciones (2 por cada una de las sentencias **if** multiplicadas por 3 derivadas de la primitiva **return**) las cuales al concatenarlas con las anteriores suman un total de 24. Junto con las 9 finales ya desarrolladas (6 en (5.17) más 3 en (5.18)) dan un total de 33 transiciones en $\mathcal{T}$ generadas a partir de procedimiento productor. La derivación de transiciones sobre el procedimiento consumidor es análoga ya que el mismo es simétrico al desarrollado. De tal manera, el conjunto de transiciones del sistema $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ que modela el problema tendrá 66 transiciones. En el apéndice 5.B (pág. 204) se muestran todas las transiciones obtenidas.

Como puede notarse, el proceso de generación manual de transiciones a partir de un monitor dado resulta extenso y engorroso (en los problemas analizados en la sección 5.8 se llegan a 740 transiciones). Es por ello que parte del prototipo de software que implementa nuestro método consiste en la generación mecánica de las mismas, por aplicación de las reglas desarrolladas en esta sección. Las transiciones mostradas en aquel apéndice fueron obtenidas por este prototipo.□

Como ya mencionamos, a cada guarda de las operaciones **await** se le asocia una variable de condición en la implementación del monitor. En el sistema de transiciones que lo modela, cada una estará asociada a una locación. Además tendremos una locación extra e para los procesos intentando invocar un procedimiento. Por lo tanto, si en el monitor aparecen m operaciones **await**, el conjunto de locaciones será:

$$\mathcal{L} \ \doteq\ \{e\} \cup \{c_i \mid 1 \leqslant i \leqslant m\}\}$$

donde c_i son las variables de condición.

El conjunto de estados iniciales θ del monitor debe ser dado de forma explícita al momento de definir el problema como lo hicimos en el programa 5.3 (pág. 160). A partir del mismo y las condiciones iniciales de las colas (vacías al comienzo de la ejecución) podemos construir el conjunto de configuraciones iniciales:

$$\begin{array}{l} \Theta.e \ \doteq\ \theta \wedge \langle \forall j\ : 1 \leqslant j \leqslant m :\ w_j = 0 \wedge s_j = 0 \,\rangle\ , \\ \Theta.c_i \doteq\ \mathsf{false} \quad \text{si} \quad 1 \leqslant i \leqslant m\ . \end{array}$$

Para finalizar esta sección, veremos que el sistema de transiciones que modela la implementación del monitor con señalamiento automático preserva el invariante del mismo (dado en (5.6), pág. 160).

Teorema 5.4
Sea un monitor con señalamiento automático que verifica el invariante I a partir de las condiciones de prueba

$$\begin{array}{l} \{\mathsf{true}\}\ \mathbf{enter}\ \{I\} \\ \{I\}\ \mathbf{await}.\mathsf{B}_i\ \{I \wedge \mathsf{B}_i\} \\ \{I\}\ \mathbf{return}\ \{\mathsf{false}\}\ . \end{array}$$

Sea $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ el sistema de transiciones generado a partir de la implementación del monitor. Entonces si $\Theta \subseteq I$, el invariante I es preservado en este sistema:

$$\mathsf{TS} \models \Box\varphi \quad \text{con} \quad \varphi.l \doteq I \quad (l \in \mathcal{L}) \ . \qquad \Box$$

El teorema muestra que cualquier propiedad que se deduzca sobre monitor se verificará también en el sistema de transiciones generado, con lo cual se demuestra la corrección de nuestro sistema de reglas. La demostración del mismo se encuentra en el apéndice 5.C (pág. 208).

Una vez obtenido el sistema de transiciones como se muestra en esta sección, podemos emplear las técnicas de generación de invariantes desarrolladas en el capítulo 2 y utilizadas en el capítulo anterior. La adaptación de las mismas al problema actual será desarrollada en la próxima sección.

5.5 Eliminación de señalizaciones

En la sección 5.1.1 mostramos que utilizando el estilo de codificación *signal as hint* suele suceder que algunas de las operaciones **signal** pueden ser eliminadas. Este estilo de codificación aparece en la implementación de monitores con señalamiento automático dada en la sección 5.3. En la sección actual comenzaremos a desarrollar el método para eliminar estas señalizaciones superfluas.

El método es análogo al desarrollado en el capítulo anterior (sección 4.5): a partir de la modelización de la implementación del monitor de señalamiento automático con un sistema de transiciones (sección anterior) utilizaremos la misma técnica de propagación hacia atrás (sección 2.3.2) con el fin de probar de manera automática la factibilidad de eliminación de señalizaciones superfluas. De esta forma, el método utilizado en este capítulo es básicamente el esquematizado en el programa 4.15 (pág. 137) del capítulo anterior, adaptándolo a las particularidades propias del nuevo problema a resolver. En este nuevo contexto necesitaremos definir una nueva caracterización del invariante candidato P y alguna clase de reemplazo para el invariante inicial φ_{SBD}, el cual fue de suma importancia para la aplicación factible del método. Para esta última tarea utilizaremos la generación automática de invariantes lineales presentada en el capítulo 3. A continuación se detallan los procedimiento para construir ambos predicados.

5.5.1 Invariante candidato

En nuestra utilización del método de propagación hacia atrás, el invariante candidato debe denotar una propiedad sobre el programa cuya invariancia garantice la imposibilidad de ejecución de una operación **signal** que aparezca en la implementación del monitor. En las implementaciones, estas operaciones pueden aparecer solo en dos lugares: en la implementación de la operación **await** dada en (5.11) (pág. 166), y dentro de la implementación de la primitiva **return** en (5.14) (pág. 169). En ambas ocasiones las distintas operación **signal** aparecen siempre dentro del bucle **for** en el caso que la cola de espera correspondiente no esté vacía. Además fueron reemplazas por su accionar sobre las colas (decremento e incremento de las colas de procesos esperando y señalizados respectivamente).

Supongamos que en la parte de la implementación de una operación **await**.B_i dentro de un procedimiento se quiere verificar la posibilidad de eliminación de una señalización **signal**.c_k $(k \neq i)$ dentro de aquel bucle. Una posibilidad es demostrar que antes de la ejecución del bucle, sea invariante que la cola w_k es vacía $(w_k = 0)$. En este caso la sentencia condicional dentro del bucle elegirá la segunda opción, ejecutando siempre **skip**. En la siguiente implementación de la operación **await** se anotó esta condición:

$$
\begin{array}{lll}
\textbf{await}.\mathsf{B}_i & \doteq & \underline{\textbf{if}}\ \ \mathsf{B}_i \mapsto \textbf{skip} \\
 & & \Box\ \neg\mathsf{B}_i \mapsto \\
 & & \quad \{w_k = 0?\} \\
 & & \quad \textbf{for}\ (j = 1, \cdots, m \wedge j \neq i) \\
 & & \quad\quad \underline{\textbf{if}}\ w_j > 0 \mapsto w_j, s_j := w_j - 1, s_j + 1 \\
 & & \quad\quad \Box\ \ w_j = 0 \mapsto \textbf{skip} \\
 & & \quad\quad \underline{\textbf{fi}}; \\
 & & \quad \textbf{waitLoop}_i \\
 & & \underline{\textbf{fi}}
\end{array}
$$

Para poder eliminar la señalización **signal**.c_k en la anterior implementación esta condición es demasiado fuerte. Una segunda posibilidad es que siempre que se señalice a procesos esperando en la cola w_k, si la condición B_k se verifica entonces haya procesos señalizados con anterioridad $(s_k > 0)$. Esto quiere decir que no es necesario ejecutar la señalización, ya que la posibilidad de activar procesos esperando en la variable de condición ya esta garantizada. Por lo tanto, si antes del bucle **for** se verifica de manera invariante $\mathsf{B}_k \Rightarrow s_k > 0$ entonces la señalización es superflua. Junto con el invariante candidato anterior incluiremos esta segunda opción en el siguiente programa:

$$
\begin{array}{lll}
\textbf{await}.\mathsf{B}_i & \doteq & \underline{\textbf{if}}\ \ \mathsf{B}_i \mapsto \textbf{skip} \\
 & & \Box\ \neg\mathsf{B}_i \mapsto \\
 & & \quad \{w_k = 0\ \vee\ \mathsf{B}_k \Rightarrow s_k > 0?\} \\
 & & \quad \textbf{for}\ (j = 1, \cdots, m \wedge j \neq i) \\
 & & \quad\quad \underline{\textbf{if}}\ w_j > 0 \mapsto w_j, s_j := w_j - 1, s_j + 1 \\
 & & \quad\quad \Box\ \ w_j = 0 \mapsto \textbf{skip} \\
 & & \quad\quad \underline{\textbf{fi}}; \\
 & & \quad \textbf{waitLoop}_i \\
 & & \underline{\textbf{fi}}
\end{array}
$$

De esta forma, propagando hacia atrás con el transformador wlp el predicado $w_k = 0\ \vee\ \mathsf{B}_k \Rightarrow s_k > 0$ dentro del sistema de transiciones, se encontrará el predicado P cuya invariancia garantiza la posibilidad de eliminación de la operación **signal**. Esta propagación para encontrar el invariante candidato es análoga a la explicada en el capítulo anterior (sección 4.5). El único problema que surge en este nuevo contexto es que el sistema de transiciones actual es derivado a partir de las reglas desarrolladas en la sección anterior sobre cada problema de monitores en particular y no en la forma de una enumeración general como se hizo con las implementaciones SBD.

Con el fin de sistematizar la propagación de la condición de eliminación de una señalización, dejaremos la aparición particular de **await**.B_i (donde aparece

la señalización a eliminar) tal cual se encuentra en el monitor con señalamiento automático (sin implementar) y agregaremos las siguientes reglas específicas para esta operación:

$$\begin{array}{lcl} \textbf{await}.\mathsf{B}_i & \vdash & (\bullet,\ \square\ \mathsf{B}_i \mapsto \textbf{skip},\ \bullet) \\ \textbf{await}.\mathsf{B}_i & \vdash & (\bullet,\ \square\ \neg\mathsf{B}_i \mapsto \textbf{skip},\ \bar{c}\,)\ . \end{array} \tag{5.20}$$

La primer regla modela el caso donde la condición de espera se satisface (el proceso continúa con su ejecución). La segunda regla crea una locación nueva antes de la ejecución del bucle **for** la cual será el punto de partida para propagar la condición $w_k = 0\ \vee\ \mathsf{B}_k \Rightarrow s_k > 0$. Con las nuevas reglas incluidas se genera un conjunto de transiciones $\mathcal{T}'$ temporal (solo para encontrar el invariante candidato P) y se propaga la condición sobre todas las transiciones con locación de llegada $\bar{c}$, obteniéndose así el invariante candidato:

$$P.l\ \doteq\ \langle \bigcap s\ : \mathcal{T}'.l.s.\bar{c} :\ \text{wlp}.s.(w_k = 0\ \vee\ \mathsf{B}_k \Rightarrow s_k > 0)\,\rangle,\quad l \in \mathcal{L}\ . \tag{5.21}$$

La demostración de invariancia de este predicado será condición suficiente para la eliminación de una señalización $\textbf{signal}.c_k$ (con $k \in \{1, \cdots, m\} - \{i\}$) en la implementación de la operación $\textbf{await}.\mathsf{B}_i$.

Resumiendo, el método de eliminación de señalizaciones dentro de la implementación de una operación $\textbf{await}.\mathsf{B}_i$ en particular, comenzará generando un nuevo sistema de transiciones (dejando sin implementar la operación y agregando las reglas anteriores). Eligiendo una señalización $\textbf{signal}.c_k$ dentro de esta operación, se propagará hacia atrás, sobre el nuevo sistema de transiciones, el predicado $w_k = 0\ \vee\ \mathsf{B}_k \Rightarrow s_k > 0$, como se indica en la ecuación anterior. Si el calculo de punto fijo converge a un predicado más débil que el conjunto de configuraciones iniciales Θ, entonces la señalización puede ser eliminada. Este proceso es repetido para cada implementación de una operación $\textbf{await}.\mathsf{B}_i$ (generando un sistema de transiciones nuevo para cada una) y sobre cada señalización contenida en ella. A continuación veremos un ejemplo de este proceso.

Ejemplo 5.5 (Invariante candidato en await)
En el ejemplo 5.3 (pág. 171) vimos como generar el conjunto de transiciones $\mathcal{T}$ a partir la implementación del monitor Productor/Consumidor. En aquel monitor aparecen solo dos operaciones **await** (una en el productor y otra en el consumidor) cuyas implementaciones contienen un sola señalización (a procesos ejecutando distintos procedimientos). En este ejemplo obtendremos el invariante candidato correspondiente a la señalización que se produce dentro de la operación en el procedimiento productor. Para el caso en cuestión, la condición para su eliminación será:

$$w_2 = 0\ \vee\ (n > 0 \Rightarrow s_2 > 0)\ \ ,$$

lo cual significa que la señalización al consumidor puede eliminarse si, de manera invariante, no hay procesos esperando o si la condición de consumo se cumple entonces algún proceso consumidor ya fue señalizado.

Como primera medida, reescribiremos el procedimiento productor sin implementar su operación $\textbf{await}.n < N$ e implementando solo la primitiva **return** (programa 5.6, pág. 179).

Programa 5.6 Implementación del procedimiento productor

```
procedure Prod()
   enter;
   await.(n < N);
   n := n + 1;
   if w1 > 0 ↦ w1, s1 := w1 − 1, s1 + 1
   ▯ w1 = 0 ↦ skip
   fi;
   if w2 > 0 ↦ w2, s2 := w2 − 1, s2 + 1        return
   ▯ w2 = 0 ↦ skip
   fi;
   return
end
```

Antes de generar el nuevo sistema de transiciones $\mathcal{T}'$, agregaremos las dos reglas nuevas, esquematizadas en (5.20), correspondientes a esta operación:

$$\begin{array}{lcl} \textbf{await}.(n < N) & \vdash & (\bullet,\ \Box\, n < N \mapsto \textbf{skip},\ \bullet) \\ \textbf{await}.(n < N) & \vdash & (\bullet,\ \Box\, \neg n < N \mapsto \textbf{skip},\ \bar{c}\,)\ . \end{array}$$

Con estas nuevas reglas generaremos las transiciones de la misma forma que en el ejemplo 5.3.

La sentencia **enter** genera la misma transición que en ejemplo anterior ((5.15) en pág. 171), la cual puede ser concatenada (por la regla *Seq1*, pág. 164) con las generadas por las dos reglas nuevas:

$$\begin{array}{l} (e,\ \Box\, n < N \mapsto \textbf{skip},\ \bullet)\ , \\ (e,\ \Box\, \neg n < N \mapsto \textbf{skip},\ \bar{c})\ . \end{array}$$

La primer transición será concatenada con las generadas desde la asignación $n := n + 1$, de igual forma que en el ejemplo anterior. La segunda formará parte del sistema de transiciones $\mathcal{T}'$ (se preserva por aplicación de la regla *Seq2*). Esta será la única con locación de llegada $\bar{c}$ ya que la regla agregada que la genera solo puede aplicarse a esta parte del procedimiento. Por lo tanto, según la ecuación (5.21), el invariante candidato será:

$$\begin{array}{rcl} P.e & \doteq & \langle \bigcap s\ : \mathcal{T}'.e.s.\bar{c} :\ \text{wlp}.s.(w_2 = 0\ \vee (n > 0 \Rightarrow s_2 > 0))\,\rangle \\ & = & \text{wlp}.(\Box\, \neg n < N \mapsto \textbf{skip}).(w_2 = 0\ \vee (n > 0 \Rightarrow s_2 > 0)) \\ & = & \neg n < N \Rightarrow (w_2 = 0\ \vee (n > 0 \Rightarrow s_2 > 0)) \\ P.c_1 & \doteq & \mathsf{true} \\ P.c_2 & \doteq & \mathsf{true} \end{array}$$

Este resultado en la locación de entrada e indica que si de manera invariante cada vez que se invoca un procedimiento del monitor, si la condición de producción no se cumple, entonces o no hay procesos consumidores esperando o si la condición de consumo se cumple entonces ya se señalizó con anterioridad. La invariancia implica que si un proceso invoca el procedimiento productor sin cumplir su condición, entonces no necesita señalizar un consumidor. □

Para comprobar la posibilidad de eliminación de las señalizaciones que aparecen en las implementación de la primitiva **return** (5.14) se procede de la misma manera. La condición para eliminar una operación **signal**.c_k en la implementación de la primitiva (al final de un procedimiento) es idéntica a la vista anteriormente por los mismos argumentos:

$$
\begin{array}{rcl}
\textbf{return}' & \doteq & \{w_k = 0 \ \vee \ \mathsf{B}_k \Rightarrow s_k > 0?\} \\
& & \textbf{for}\ (j = 1, \cdots, m) \\
& & \quad \underline{\textbf{if}}\ w_j > 0 \mapsto w_j, s_j := w_j - 1, s_j + 1 \\
& & \quad \square\ \ w_j = 0 \mapsto \textbf{skip} \\
& & \quad \underline{\textbf{fi}}; \\
& & \textbf{return}
\end{array}
$$

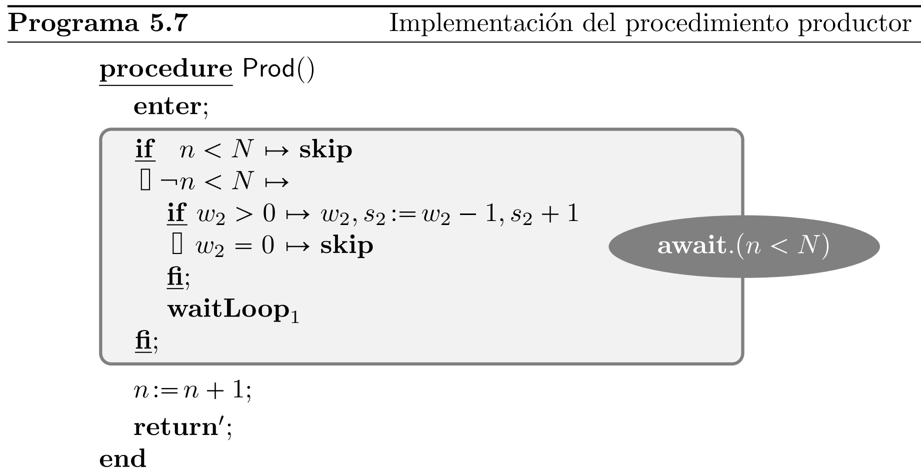
donde hemos primado esta aparición particular de la primitiva **return** (dentro de la cual se intenta eliminar la señalización) para diferenciarla de las que aparecen en otros procedimientos. De forma análoga que con la operación **await**, dejamos sin implementar esta primitiva en particular e incluimos solo una nueva reglas para su aparición en la finalización de un procedimiento:

$$
\textbf{return}' \ \ \vdash \ \ (\bullet,\ \textbf{skip},\ \bar{c}) \tag{5.22}
$$

En este caso solo necesitamos una regla extra ya que la condición a verificar se encuentra al principio de la implementación y no dentro de una sentencia condicional como en el caso anterior.

Generando el nuevo conjunto de transiciones, si queremos probar la posibilidad de eliminación de una señalización **signal**.c_k ($k \in \{1, \cdots, m\}$) dentro de la implementación de esta primitiva, propagamos la anterior condición sobre la transiciones cuya locación de llegada es $\bar{c}$ de la misma forma que se muestra en (5.21).

Ejemplo 5.6 (Invariante candidato en return)
En este ejemplo obtendremos un invariante candidato para la eliminación de una señalización en la implementación de la primitiva **return** en el procedimiento productor. El programa con esta primitiva sin implementar (señalada con **return′**) será:



Programa 5.7 Implementación del procedimiento productor

```
procedure Prod()
   enter;
   if   n < N ↦ skip
   □ ¬n < N ↦
      if w2 > 0 ↦ w2, s2 := w2 − 1, s2 + 1
      □ w2 = 0 ↦ skip
      fi;
      waitLoop1
   fi;
   n := n + 1;
   return′;
end
```

Para generar el conjunto de transiciones $\mathcal{T}'$ agregaremos la regla (5.22). La misma se utiliza solo al final del programa y es la única que tiene como locación de llegada a $\bar{c}$ generando la transición:

$$(\bullet,\ \textbf{skip},\ \bar{c})\ \ .$$

Esta será concatenada con las generadas desde el comienzo del programa hasta la implementación de la operación **await**.$(n < N)$ y posean locación de llegada auxiliar "$\bullet$" (por aplicación de la regla *Seq1*), las cuales ya fueron generadas en el ejemplo 5.3 (pág. 171) en (5.19):

$$\begin{array}{l}(e,\ \square\, n < N \mapsto n\!:=\!n+1,\ \bullet)\ \ ,\\ (c_1,\ \square\, n < N \mapsto n\!:=\!n+1,\ \bullet)\ \ .\end{array}$$

Por aplicación de la regla *Seq1* estas últimas con la anterior generan las transiciones:

$$\begin{array}{l}(e,\ \square\, n < N \mapsto n\!:=\!n+1,\ \bar{c})\ \ ,\\ (c_1,\ \square\, n < N \mapsto n\!:=\!n+1,\ \bar{c})\ \ ,\end{array}$$

las cuales serán las únicas pertenecientes a $\mathcal{T}'$ con locación de llegada $\bar{c}$, por lo que contribuirán al cálculo del invariante candidato. En el caso que intentemos eliminar la señalización a un productor desde la implementación de esta primitiva **return** el invariante candidato será:

$$\begin{array}{rl}P.e & \doteq\ \langle \bigcap\ s\ :\mathcal{T}'.e.s.\bar{c}:\ \text{wlp}.s.(w_1 = 0\ \vee\ (n < N \Rightarrow s_1 > 0))\,\rangle\\ & = \text{wlp}.(\square\, n < N \mapsto n\!:=\!n+1).(w_1 = 0\ \vee\ (n < N \Rightarrow s_1 > 0))\\ & = n < N \Rightarrow (w_1 = 0\ \vee\ (n+1 < N \Rightarrow s_1 > 0))\\ & = n \leqslant N-2 \Rightarrow (w_1 = 0\ \vee\ s_1 > 0)\\ P.c_1 & \doteq\ \langle \bigcap\ s\ :\mathcal{T}'.c_1.s.\bar{c}:\ \text{wlp}.s.(w_1 = 0\ \vee\ (n < N \Rightarrow s_1 > 0))\,\rangle\\ & = \text{wlp}.(\square\, n < N \mapsto n\!:=\!n+1).(w_1 = 0\ \vee\ (n < N \Rightarrow s_1 > 0))\\ & = n \leqslant N-2 \Rightarrow (w_1 = 0\ \vee\ s_1 > 0)\\ P.c_2 & \doteq\ \mathsf{true}\ \ .\end{array}$$

El invariante candidato para eliminar la otra señalización (a un consumidor) se propaga de la misma forma la condición $w_2 = 0\ \vee (n > 0 \Rightarrow s_2 > 0)$:

$$\begin{array}{rl}P.e & \doteq\ \langle \bigcap\ s\ :\mathcal{T}'.e.s.\bar{c}:\ \text{wlp}.s.(w_2 = 0\ \vee\ (n > 0 \Rightarrow s_2 > 0))\,\rangle\\ & = \text{wlp}.(\square\, n < N \mapsto n\!:=\!n+1).(w_2 = 0\ \vee\ (n > 0 \Rightarrow s_2 > 0))\\ & = n < N \Rightarrow (w_2 = 0\ \vee\ (n+1 > 0 \Rightarrow s_2 > 0))\\ P.c_1 & \doteq\ \langle \bigcap\ s\ :\mathcal{T}'.c_1.s.\bar{c}:\ \text{wlp}.s.(w_2 = 0\ \vee\ (n > 0 \Rightarrow s_2 > 0))\,\rangle\\ & = \text{wlp}.(\square\, n < N \mapsto n\!:=\!n+1).(w_2 = 0\ \vee\ (n > 0 \Rightarrow s_2 > 0))\\ & = n < N \Rightarrow (w_2 = 0\ \vee\ (n+1 > 0 \Rightarrow s_2 > 0))\\ P.c_2 & \doteq\ \mathsf{true}\ \ .\end{array}$$

Notar que hemos utilizado el mismo sistema de transiciones $\mathcal{T}'$ para generar ambos invariantes candidatos. □

5.5.2 Invariante inicial

En el capítulo anterior la utilización del invariante inicial φ_{SBD} fue necesario para lograr la convergencia del método de propagación hacia atrás. El mismo fue utilizado como suposición de las pruebas de las implicaciones utilizadas en el método y para simplificar los invariantes candidatos intermedios. En todos los casos de prueba se confirmó que sin su utilización el método no puede decidir la invariancia del invariante candidato. En el contexto actual de implementación de monitores, el espacio de estados y los sistemas de transiciones resultantes son de mayor tamaño[3], comparando las modelizaciones para los mismos problemas. El aumento en el tamaño del espacio de estados se debe a la inclusión de más contadores auxiliares en la modelización. En el caso de implementaciones SBD solo utilizamos un contador b_i por cada semáforo s_i (distinto al neutral s_m). En el caso de monitores, cada variable de condición c_i tiene asociada dos contadores, uno para el tamaño de la cola de procesos en espera w_i y otro para el tamaño de la cola de señalizados s_i. Esto duplica la cantidad de variables auxiliares y hace más complejo el comportamiento operacional del sistema dificultando su análisis.

Por otro lado, al modelar los mismos problemas, la cantidad de transiciones aumenta drásticamente. Esto fenómeno puede ser comprobado analizando las implementaciones (5.11) (pág. 166) y (5.14) (pág. 169) de las cuales se derivan las transiciones (por aplicación de las reglas allí desarrolladas). En ambas implementaciones el bucle **for** se despliega en $m-1$ y m composiciones secuenciales de sentencias condicionales con dos guardas. Aplicando las reglas *Seq1* (5.9) e *If1* (5.10) (pág. 164 y 165) se obtienen del orden de 2^m transiciones tanto para cada aparición de operaciones **await** como primitivas **return** al final de cada procedimiento. Este crecimiento exponencial en la cantidad de transiciones generadas es resultado del comportamiento operacional de los monitores no bloqueantes (NPNB) donde un señalamiento no implica la ejecución inmediata del proceso señalizado: las señalizaciones modeladas son realizadas dentro del bucle **for** de forma independiente al próximo proceso activo que comience la ejecución. Justamente, este comportamiento hace la diferencia con los sistemas de transiciones generados desde implementaciones SBD. Allí, una operación V (análoga a una señalización) es seguida por una operación P sobre el mismo semáforo y en el nuevo proceso activo como indica la *regla del dominó* (pág. 119). Notar además, que para modelar este nuevo comportamiento fue necesario agregar los dos contadores produciendo en consecuencia, como se menciona anteriormente, el aumento en el tamaño del espacio de estados.

El no determinismo generado por este tipo de monitor produce también mayor número de transiciones en el caso que una guarda de una operación **await** no se cumpla. En este caso, por aplicación de las regla *WakeEnter* (5.12) y *WakeWait* (5.13), se puede despertar cualquier proceso entrante o esperando por una condición, a diferencia de las implementaciones SBD donde solo se despierta un proceso entrante.

Por lo dicho anteriormente, es clara la necesidad de poder contar con un invariante inicial para simplificar el cálculo del punto fijo. A diferencia de nuestro trabajo anterior, no contamos a priori con un invariante del tipo φ_{SBD} producido de forma directa por la implementación del monitor. Para obtenerlo utilizamos

[3]Los detalles de este fenómeno serán mostrados en la sección 5.8 donde se comparan estos tamaños para los problemas de Productor/Consumidor y para Lectores/Escritores.

la técnica de interpretación abstracta desarrollada en el capítulo 3: a partir del sistema de transiciones $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ dado en la sección 5.4 obtenemos uno abstracto $\mathsf{TS}^\sharp \doteq (\mathcal{L}, \mathcal{S}^\sharp, \mathcal{T}^\sharp, \Theta^\sharp)$ como se indica en la sección 3.3.5 (pág. 99) y generamos un invariante realizando propagación hacia adelante desde el conjunto de configuraciones iniciales abstracto, como se indica en aquel capítulo. Sea I_{POL} el invariante (concretizado) generado de esta forma. Definiremos entonces el invariante inicial como:

$$\varphi_{MON}.l \doteq I_{POL}.l \wedge \langle \forall j : 1 \leqslant j \leqslant m : w_j \geqslant 0 \wedge s_j \geqslant 0 \rangle \tag{5.23}$$

con $l \in \{e, c_1, \cdots, c_m\}$. Notar que agregamos al invariante lineal las condiciones sobre las longitudes de las colas de procesos (son siempre positivas).

De esta manera, el método de propagación hacia atrás en el programa 4.15 (pág. 137) se modifica para utilizar este invariante:

Programa 5.8 Propagación hacia atrás con simplificación

es_invariante (P, Θ, φ_{MON}: $\mathsf{Pred}_{\mathcal{L}}$; $\mathcal{T}$: $\mathsf{Tran}_{\mathcal{L},\mathcal{S}}$) : Bool
$\mathcal{B}_{\mathcal{T},P} := \langle \lambda X \bullet \mathbf{simplificar}.\varphi_{MON}.(P \cap \mathrm{WLP}.\mathcal{T}.X) \rangle$;
$B_k := \mathsf{True}$;
do $\neg(\varphi_{MON} \vdash \neg(\Theta \subseteq B_k)) \wedge \neg(\varphi_{MON} \vdash B_k \subseteq \mathcal{B}_{\mathcal{T},P}.B_k)$
$B_k := \mathcal{B}_{\mathcal{T},P}.B_k$
od;
return $\varphi_{MON} \vdash \Theta \subseteq B_k$

donde el parámetro P (invariante candidato) es obtenido como se indica en la sección anterior.

5.6 Mejoras del método

Con el fin de agilizar el cálculo del punto fijo se agregaron algunas mejoras al método de propagación hacia atrás. Las estrategias de simplificación de fórmulas descriptas en el capítulo anterior (sección 4.6.4) se reutilizaron tal cual fueron descriptas en aquel trabajo, con el agregado que también fueron empleadas para simplificar el sistema de transiciones: toda transición $(l, \Box\, B \mapsto S, l') \in \mathcal{T}$ es reemplazada por

$$(l, \Box\, \mathbf{simplificar}'.(\varphi_{MON}.l).B \mapsto S, l') \ ,$$

donde la función **simplificar′** es la aplicación puntual de la anterior a solo un elemento de Pred. La estrategia fue aplicada en el ejemplo 5.3 (pág. 171) donde además se transformaron las asignaciones secuenciales en los comandos guardados de las transiciones por asignaciones múltiples y se eliminaron las asignaciones superfluas (como se indica en [BAW98, teoremas 5.3 y 5.4]). Todas estas transformaciones fueron aplicadas a nuestro método.

En el capítulo anterior se elaboraron también estrategias para la eliminación de transiciones (sección 4.6.2). En el problema actual no es posible enumerar a priori estas eliminaciones de manera general como se hizo en aquella ocasión,

debido a la falta de estructura sintáctica regular, como se menciona en la sección 5.4. Es por ello que para cada problema en particular se utilizó también el invariante φ_{MON} para eliminar transiciones. Tomando la transición anterior, si para todo estado definido por le invariante la guarda anterior no se satisface:

$$\vdash \varphi_{MON}.l \subseteq \neg B \ ,$$

entonces la transición puede ser eliminada. La demostración de esta implicación fue realizada utilizando los demostradores externos de la misma forma que para las implicaciones en el cálculo del punto fijo.

Esta última estrategia reduce la cantidad de transiciones solo si existen transiciones superfluas en el sistema generado inicialmente. Otra mejora en el mismo sentido, fue la de eliminar de forma progresiva las reglas de generación que contienen las señalizaciones probadas superfluas. Esto es, para cada señalización dentro de la implementación del monitor (en las implementaciones de las operaciones **await** y las primitivas **return**), si se demuestra que es superflua, entonces se la elimina del rango del bucle **for** en (5.11) y (5.14) (pág 166 y 169). A partir de esta modificación en la implementación se genera un nuevo sistema de transiciones. El proceso es aplicado cada ves que se pruebe la eliminación de una señalización.

De la misma manera que con las transiciones, el invariante φ_{MON} también fue refinado utilizando los resultados de etapas anteriores en el proceso global de búsqueda de señalizaciones superfluas. Como ya señalamos en el programa 5.8, se utilizó el invariante inicial φ_{MON} de igual forma que en el capítulo anterior (sección 4.6) pero fortaleciéndolo de manera incremental: para cada señalización a eliminar dentro de los procedimientos del monitor, si el programa 5.8 converge a un invariante (se puede eliminar la señalización), entonces será utilizado para fortalece φ_{MON}. Si el programa termina en esta situación, en B_k se almacena este invariante y a partir del mismo se reemplaza φ_{MON} por $\varphi_{MON} \cap B_k$, el cual será utilizado en las pruebas de eliminación sucesivas. Esto permite aumentar las suposiciones en las pruebas de las implicaciones de aquel programa, disminuyendo el crecimiento de los predicados intermedios y agilizando los cálculos de puntos fijos futuros. Todas estas mejoras fueron agregadas en la implementación del método, el cual será descripto en la sección siguiente.

5.7 Implementación

Con el fin de probar el método propuesto sobre distintos problemas, se desarrollo un prototipo de software que implementa un monitor con señalamiento automático con uno de señalamiento explícito, eliminando la señalizaciones innecesarias. El software reutiliza gran parte del desarrollado para el trabajo anterior, principalmente el programa de cálculo de punto fijo y los métodos de pruebas y simplificaciones utilizando los probadores externos y las tácticas propias. El mismo fue también escrito el lenguaje ML, en su implementación *Standard ML of New Jersey* [Sta06], utilizando como procesos externos los probadores de teoremas CVC3 e Isabelle/HOL.
Resumiendo, el prototipo de software implementa las siguientes etapas:

Inicialización. A partir de un monitor con señalamiento automático se procede a ejecutar los siguientes pasos:

1. Se genera el sistema de transiciones inicial por aplicación de las reglas desarrolladas en la sección 5.4. En este paso también se transforman las asignaciones secuenciales generadas por asignaciones múltiples y se eliminan asignaciones superfluas.
2. Con este sistema de transiciones se obtiene uno abstracto para utilizar el método de generación de invariantes lineales por interpretación abstracta en el dominio de poliedros convexos (capítulo 3). Con el invariante así obtenido y agregando las condiciones sobre las colas, se obtiene el invariante inicial φ_{MON} de la ecuación (5.23).
3. Con este invariante se eliminan transiciones y se simplifican sus guardas, como se indica en la sección 5.6.

Esta etapa devuelve como resultado un sistema de transiciones simplificado (que modela la implementación del monitor con señalamiento automático) y el invariante inicial φ_{MON}.

Eliminación de señalizaciones. Para cada operación **signal** que aparece en la implementación del monitor se ejecutan los siguientes pasos:

1. Se obtiene un invariante candidato P desde un nuevo sistema de transiciones generado, como se indica en la sección 5.5.1.
2. Con el sistema de transiciones (resultado de la etapa anterior), el invariante inicial φ_{MON} y el invariante candidato P se intenta probar la invariancia de este último. Este paso fue implementado reutilizando el programa que implementa el cálculo del punto fijo por propagación hacia atrás, en el programa 4.15 del capítulo anterior.
3. Según este último resultado, si la señal en cuestión es superflua se la elimina de la implementación del monitor, generando un nuevo sistema de transiciones para verificar la posibilidad de eliminación de la próxima señalización (sección 5.6).
4. En el mismo caso que la señalización actual sea superflua, el invariante generado en el cálculo de punto fijo es utilizado para fortalecer el invariante φ_{MON}. Este nuevo invariante fortalecido será utilizado como invariante inicial para la próxima señalización a tratar.

En la figura 5.1 (pág. 202) se esquematizan estas etapas y sus dependencias con módulos externos. Ambas etapas fueron incluidas dentro de mismo módulo ML para poder compartir las funciones que implementan la generación de transiciones a partir de reglas y el uso de los probadores y simplificadores de formulas de manera sencilla. A continuación profundizaremos este diseño.

5.7.1 Etapa de inicialización

Esta etapa tiene como entrada el monitor con señalamiento automático a tratar. El mismo es ingresado al sistema como un archivo de texto respetando una gramática similar a la del lenguaje C pero reducida y adaptada para especificar el problema. En la sección 5.8 se mostrarán ejemplos de la misma. Esta entrada es interpretada en una serie de tipos de datos ML que la representa.

Con esta entrada elaborada, el prototipo genera el sistema de transiciones por aplicación de las reglas, desplegando antes las implementaciones de las operaciones **await** y primitivas **return** como se indica en la sección 5.4. Además se eliminan las asignaciones superfluas en los comandos guardados y se transforman las asignaciones secuenciales en asignaciones múltiples como se indica en [BAW98, teoremas 5.3 y 5.4].

La próxima parte de la etapa consiste en encontrar el invariante inicial φ_{MON}. Para ello se utilizó el módulo de interpretación abstracta en poliedros convexos donde genera el sistema de transiciones abstracto y se implementa el método de propagación hacia adelante en el programa 3.4 (pág. 101). Este último hace uso de la librería para la manipulación de poliedro Polka [HPR97]. La librería se encuentra escrita en lenguaje C por lo cual hubo que hacer una interfase entre este lenguaje y ML utilizando la librería NLFFI [Blu01]. A partir del invariante lineal concretizado generado por este módulo, más las condiciones sobre los tamaños de las colas (5.23) se construye el invariante inicial sobre el sistema de transiciones.

La próxima parte de esta etapa consiste en simplificar el sistema de transiciones utilizando el invariante inicial, como se indica en la sección 5.6. De esta manera, se simplifican las guardas y se eliminan transiciones superfluas manteniendo como suposición aquel invariante. Para implementar este proceso se reutilizaron las estrategias de prueba y simplificación realizadas con los probadores externos CVC3 [CVC08] e Isabelle/HOL [Pau94]. Para esta ocasión se implementó un módulo que agrupa estas funcionalidades, las cuales también serán utilizadas en la etapa siguiente.

Esta etapa arroja como resultado el sistema de transiciones y el invariante inicial, los cuales será utilizado en la etapa siguiente, como se indica en la figura 5.1 (pág. 202).

5.7.2 Etapa de eliminación de señalizaciones

Esta etapa realiza las pruebas que posibilitan la eliminación de las operaciones **signal** en la implementación del monitor de entrada. Como ya se mencionó, el método consiste en la búsqueda de invariantes que garanticen esta eliminación, refinando el sistema de transiciones y el invariante inicial a partir de cada invariante encontrado: procesando secuencialmente cada señalización, si resulta superflua, el invariante que prueba este hecho es utilizado para fortalecer φ_{MON} y para eliminar las transiciones cuyas guardas no sean satisfactibles en el conjunto de configuraciones definido por el mismo (sección 5.6).

Para cada señalización a tratar se genera un sistema de transiciones temporal con el fin de producir el invariante candidato P, como se indica en la sección 5.5.1. La parte de generación de estas transiciones es el utilizado en la etapa anterior incluyendo las reglas (5.20) y (5.22) definidas en aquella sección.

El cálculo del punto fijo a partir del método de propagación hacia atrás es el mismo que el empleado en el capítulo anterior. Este cálculo también permite detectar la imposibilidad de eliminación de la señal, en el caso que el conjunto de configuraciones iniciales Θ no este contenido en algún elemento de la cadena generada por el programa 5.8. Como ya se mencionó, para esta ocasión se implementó un módulo que agrupa las técnicas de prueba y simplificación utilizadas en el cálculo anterior (y en la etapa anterior), como se indica en la figura 5.1 (pág. 202).

Al final de la etapa se procede a generar la implementación del monitor de entrada con un monitor con señalamiento explícito sin las señalizaciones superfluas. Para ello se reescribe el mismo monitor de entrada, desplegando las implementaciones de las operaciones **await** y las primitivas **return** (sección 5.3) y eliminando aquellas operaciones **signal**. El resultado es mostrado en forma de texto como un programa monitor con la misma gramática estilo C del archivo de entrada. En la próxima sección mostraremos ejemplos de estas salidas.

5.8 Resultados

El prototipo de software fue probado sobre diversos problemas de la literatura. El mismo fue ejecutado sobre una computadora con arquitectura PC, procesador de 2GHz. y 2Gb de memoria RAM. A la elección de casos de prueba del capítulo anterior (sección 4.8) fueron agregados nuevos ejemplos. A continuación se detallan los resultados obtenidos comenzando con el ejemplo más simple.

5.8.1 Semáforos generales

Comenzaremos con el problema de implementación de un semáforo general (tratado en la sección 4.8.3 del capítulo anterior) con monitores. El archivo de entrada con el monitor de señalamiento automático que resuelve este problema es:

```
Monitor semaphore {
int s;

    semaphore(){ /*@ require: s == 0 @*/ }

    p(){
      await (s > 0);
      s = s - 1;
    }

    v(){ s = s + 1; }
}
```

Como se puede observar, se ha incluido un procedimiento con el mismo nombre del monitor. Esta inclusión en la gramática del archivo de entrada sirve para denotar la inicialización de las variables del monitor. Dentro de este método se agrega la directiva `/*@ require: s == 0 @*/` para señalar que inicialmente la variable `s` tiene el valor 0. Los demás procedimientos son los correspondientes a las operaciones P y V sobre el semáforo (equivalentes a las regiones críticas condicionales en el capítulo anterior, sección 4.8.3).

A continuación mostraremos la implementación con un monitor de señalamiento explícito sin simplificar (a la izquierda), y con las eliminaciones de señales que encuentra nuestro método (a la derecha).

Sin simplificar

```
Monitor semaphore {
  int s; cond c1;

    p(){
      while (!0 < s)
          wait(c1);
      s = s - 1;
      signal(c1);
    }

    v(){
      s = s + 1;
      signal(c1);
    }
}
```

Simplificado

```
Monitor semaphore {
  int s; cond c1;

    p(){
      while (!0 < s)
          wait(c1);
      s = s - 1;
    }


    v(){
      s = s + 1;
      signal(c1);
    }
}
```

El método encuentra la señalización al finalizar el procedimiento `p()` superflua en algo menos de 3 segundos, arrojando el programa de la derecha. La otra señalización (al finalizar `v()`) es necesaria ya que el procedimiento de cálculo de punto fijo llega a un elemento de la cadena que no es implicado por el conjunto de configuraciones iniciales.

Notar que hemos simplificado la implementación de la única operación **await** y las primitivas **return** gracias a que solo hay una condición de espera. Estas simplificaciones también las realiza nuestro prototipo para visualizar de manera más compacta el resultado.

El invariante inicial construido a partir de interpretación abstracta en el dominio de poliedros (arrojado también como resultado) descubre las restricciones sobre las colas y el invariante sobre la variable s (I en el capítulo anterior, sección 4.8.3) de forma automática:

$$I_{POL}.l \doteq 0 \leqslant s \wedge 0 \leqslant w_1 \wedge 0 \leqslant s_1 \quad \text{con} \quad l \in \{e, c_1\} \enspace ,$$

donde e es la cola de entrada, c_1 es la única variable de condición ligada al predicado parámetro de la operación **await** y w_1, s_1 son los tamaños de las colas de procesos asociadas. Cabe remarcar que con este resultado el invariante I es encontrado de forma totalmente automática, en contraste con el trabajo anterior donde fue necesario introducirlo en forma manual.

La cantidad de transiciones para este ejemplo comienza en 16 y termina en 12 transiciones, gracias a la eliminación de las mismas al generar el invariante que garantiza la eliminación de la señalización. Los detalles de estos resultados son expuestos en la sección 5.8.9 (pág. 199).

5.8.2 Productor/Consumidor en buffer acotado

Este problema es el mismo que el utilizado como caso de prueba en la sección 4.8.1 del capítulo anterior. En esta ocasión, el archivo de entrada es:

```
Monitor bbuffer {
int n, N;

    bbuffer(){ /*@ require: 0 < N && n == 0 @*/ }

    producer(){
        await (n<N);
        n = n + 1;
    }

    consumer(){
        await (n>0);
        n = n - 1;
    }
}
```

A continuación se muestra la implementación sin simplificar con la obtenida por el prototipo, tal cual es mostrada de forma textual:

Sin simplificar

```
Monitor bbuffer {
  int n, N; cond c1, c2;

    producer(){
      if (!n < N) {
        signal(c2)
        wait(c1);
        while (!n < N)
          wait(c1);}
      n = n + 1;
      signal(c1);
      signal(c2);
    }

    consumer(){
      if (!0 < n){
        signal(c2)
        wait(c1);
        while (!0 < n)
          wait(c2);}
      n = n - 1;
      signal(c1);
      signal(c2);
    }
}
```

Simplificado

```
Monitor bbuffer {
  int n, N; cond c1,c2;

    producer(){
      while (!n < N)
          wait(c1);
      n = n + 1;
      signal(c2);
    }

    consumer(){
      while (!0 < n)
          wait(c2);
      n = n - 1;
      signal(c1);
    }
}
```

Como puede verse, el resultado simplificado elimina las mismas señalizaciones que en el programa 5.2 (pág. 157). Efectivamente, nuestro método encuentra todas las señales superfluas y además decide que las que aparecen en el resultado final no son posibles de eliminar (Θ no implica un elemento en la cadena).

Al resolver el problema la cantidad de transiciones van de 66 al inicio del método a 48 al final. Recordemos que en el capítulo anterior, para el mismo problema hubo solo 10 a 11 transiciones, lo cual muestra el aumento en su cantidad analizado en la sección 5.5.2 (pág. 182). El tiempo total de procesamiento es de 24 segundos contra algo más de 2 segundos en el mismo problema en el capítulo anterior.

El invariante candidato encontrado por interpretación abstracta fue:

$$I_{POL}.l \doteq 0 \leqslant n \wedge n \leqslant N \wedge -N+1 \leqslant 0 \wedge 0 \leqslant w_2 \wedge 0 \leqslant w_1 \wedge 0 \leqslant s_2 \wedge 0 \leqslant s_1$$

con $l \in \{e, c_1, c_2\}$ la locación de entrada y las identificadas con las variables de condición. Notar que en este caso también es encontrado el invariante I del problema de forma totalmente automática.

5.8.3 Sincronización de Fase

El próximo problema a tratar es el denominado *sincronización de fase*. El mismo resuelve el acceso de manera sincronizada de dos clases de procesos de forma tal que la cantidad de veces que se ejecutan sean iguales. Este problema fue presentado en [BW03] donde se elaboraron manualmente varias soluciones a partir de la teoría de Owicki-Gries [OG76] utilizando semáforos.

Utilizando monitores con señalamiento automático, el problema puede ser resuelto con dos procedimientos, uno para cada clase de procesos:

```
Monitor syncphase {
int x, y;

    syncphase(){ /*@ require: x == 0 && y == 0 @*/ }

    px(){
        await (x <= y);
        x = x + 1;
    }
    py(){
        await (y <= x);
        y = y + 1;
    }
}
```

Las variables de monitor `x` e `y` cuentan la cantidad de procesos que ejecutaron los procedimientos. La operaciones bloqueantes aseguran la sincronización.

A partir de esta entrada, nuestro método encuentra en 29 segundos la siguiente implementación:

```
Monitor syncphase {
  int x, y; cond c1, c2;

    px(){
      while (!x <= y)
          wait(c1);
      x = x + 1;
      signal(c2);
    }
```

```
    py(){
      while (!y <= x)
          wait(c2);
      y = y + 1;
      signal(c1);
    }
}
```

En la solución se eliminaron todos los señalamientos correspondientes a las implementaciones de la operaciones **await** y aquellos sobre procesos de igual clase al final de cada procedimiento. Los demás señalamientos fueron probados necesarios. Los detalles de estos resultados serán expuestos en la sección 5.8.9.

El invariante I_{POL} obtenido por el prototipo captura el introducido de forma manual $x \leqslant y + 1 \wedge y \leqslant x + 1$ en [BW03]:

$$I_{POL}.l \doteq x - y - 1 \leqslant 0 \wedge y - x - 1 \leqslant 0 \wedge 0 \leqslant s_1 \wedge 0 \leqslant w_2 \wedge 0 \leqslant w_1 \wedge 0 \leqslant s_2$$

con $l \in \{e, c_1, c_2\}$.

5.8.4 Lectores y escritores

Este caso de prueba corresponde a la solución con monitores del mismo problema tratado en la sección 4.8.4 del capítulo anterior.
El monitor de entrada es:

```
Monitor rw {
int r, w;

    rw(){ /*@ require: r == 0 && w == 0 @*/ }

    request_read(){
        await (w==0);
        r = r + 1;
    }

    release_read(){
      /*@ require: r > 0 @*/
        r = r - 1;
    }

    request_write(){
        await (w==0 && r == 0);
        w = w + 1;
    }

    release_write(){
      /*@ require: w > 0 @*/
        w = w - 1;
    }
}
```

Notar que para solucionar este problema fue necesario incluir las suposiciones $r > 0$ y $w > 0$ en los procedimientos `release_read()` y `release_write()` respectivamente. Estas condiciones son requeridas ya que cualquier programa que utilice el monitor debe liberar el permiso de lectura o escritura después de haberlo adquirido. Cabe recordar que para el tratamiento del mismo problema en el capítulo anterior estas precondiciones no fueron necesarias. Esto fue debido a que en aquel caso el invariante del problema, provisto de forma manual, implica que w (cantidad de escritores) y r (cantidad de lectores) son mayores o iguales a cero, lo cual imposibilita ejecutar aquellos procedimientos sin que se cumplan las precondiciones.

Después de 54 segundos el método devuelve la siguiente implementación del monitor anterior:

```
Monitor rw {
  int r, w; cond c1, c2;

    request_read(){
      while (!w == 0)
          wait(c1);
      r = r + 1;
      signal(c1);
    }

    release_read(){
      r = r - 1;
      signal(c2);
    }

    request_write(){
      while (!( w == 0 && r == 0 ))
          wait(c2);
      w = w + 1;
    }

    release_write(){
      w = w - 1;
      signal(c1);
      signal(c2);
    }
}
```

donde se muestra la eliminación de todas las señalizaciones en las implementaciones de las operaciones **await**, dejando solo las señalizaciones necesarias para resolver el problema: a la entrada un lector solo se señaliza lectores, a la salida solo escritores y a la salida de un escritor se señaliza a ambos tipos de procesos. Estas señalizaciones fueron detectadas como necesarias por el método (por no implicación del conjunto de configuraciones iniciales).

El invariante candidato encontrado por interpretación abstracta es el siguiente:

$$I_{POL}.l \doteq 0 \leqslant w \wedge 0 \leqslant r \wedge 0 \leqslant s_1 \wedge 0 \leqslant w_1 \wedge 0 \leqslant s_2 \wedge 0 \leqslant w_2$$

con $l \in \{e, c_1, c_2\}$ la locación de entrada y las identificadas con las variables de condición. Este invariante es más débil que el invariante I en el capítulo anterior ya que este último no es convexo ($I \doteq r \geqslant 0 \wedge (w = 0 \vee (w = 1 \wedge r = 0))$).

El mismo problema fue tratado sin incluir las precondiciones a las salidas de las lecturas y escrituras. Sin ellas el método agrega solo una señalización `signal(c2)` al finalizar el procedimiento `request_read()`, con lo cual la solución encontrada sigue siendo correcta aunque no tan eficiente como la anterior. En este caso el invariante lineal fue:

$$I_{POL}.l \doteq w \leqslant 1 \wedge 0 \leqslant s_1 \wedge 0 \leqslant w_1 \wedge 0 \leqslant s_2 \wedge 0 \leqslant w_2$$

para $l \in \{e, c_1, c_2\}$. Notar que la cantidad de escritores y lectores puede ser negativa. Esto es debido a que sin la restricción en los procedimientos de salida, los mismos pueden ejecutarse indiscriminadamente, haciendo decrecer estas variables de manera arbitraria. Cabe remarcar que en el tratamiento del mismo problema en el capítulo anterior, estas variables fueron necesariamente positivas por el invariante ingresado manualmente.

5.8.5 Baño unisex

Este problema, presentado en [And91, pág. 221], es una variación del problema de lectores y escritores. El mismo consiste en resolver la situación de acceso a un baño unisex por personas de distinto género. Los hombre solo pueden utilizar el baño cuando el mismo se encuentra desocupado. Las mujeres pueden hacerlo cuando no haya hombres en el baño y haya a lo sumo N mujeres. Representando con la variable m y w la cantidad de hombres y mujeres utilizando el baño, el monitor con señalamiento automático que resuelve el problema es el siguiente:

```
Monitor bathroom {
int m, w, N;

    bathroom(){ /*@ require: m == 0 && w == 0 && N > 0 @*/ }

    man_enter(){
        await (m == 0 && w==0);
        m = m + 1;
    }

    man_exit(){ /*@ require: m > 0 @*/
        m = m - 1;
    }

    women_enter(){
        await (w < N && m==0);
        w = w + 1;
    }

    women_exit(){ /*@ require: w > 0 @*/
        w = w - 1;
    }
}
```

En el caso que un caballero desee utilizar el baño, deberá ejecutar el procedimiento `man_enter()` al entrar y `man_exit()` al salir. De forma análoga los harán las damas con los demás procedimientos. Notar las precondiciones impuestas en este sentido sobre los procedimientos `man_exit()` y `women_exit()`.

Después de 67 segundos de procesamiento el sistema arroja el siguiente resultado:

```
Monitor bathroom {
  int m, w, N; cond c1, c2;

    man_enter(){
      while (!( m == 0 && w == 0 ))
          wait(c1);
      m = m + 1;
    }

    man_exit(){
      m = m - 1;
      signal(c1);
      signal(c2);
    }

    women_enter(){
      while (!( w < N && m == 0 ))
          wait(c2);
      w = w + 1;
      signal(c2);
    }

    women_exit(){
      w = w - 1;
      signal(c1);
      signal(c2);
    }
}
```

La misma muestra que a la salida de hombres o mujeres es necesario señalizar todos los tipos de procesos, mientras que se eliminan todas las señalizaciones de hombres entrando y las que realizan las mujeres a los hombres intentando entrar en el momento que ellas ocupan el baño. Todas las señalizaciones necesarias fueron detectadas en el cálculo de punto fijo.

Algunas variaciones sobre este problema fueron también implementadas. Al final de esta sección, en la tabla 5.1 (pág. 200) se muestran los resultados.

5.8.6 Barrera

Un problema interesante es la implementación de una sincronización en barrera [And91, pág. 120]. El mismo consiste de una cantidad no acotada de procesos que intentan ejecutar alguna acción en conjunto pero de forma sincronizada. Se requiere además que haya exactamente N procesos ejecutando la acción al

mismo tiempo. Para lograrlo se programa una barrera de sincronización que es ejecutada antes de intentar llevar a cabo la acción. Si hay N que pasaron la barrera, los procesos nuevos deben que esperar en la barrera hasta su finalización. Si hay menos de N esperando en la barrera, todos ellos deberán esperar a que se complete esta cantidad de procesos en espera, para poder ejecutar la acción en conjunto. Una solución posible es mediante el siguiente monitor:

```
Monitor barrier {
int i, j, N;

    barrier(){ /*@ require: i == 0 && j == 0 && 0 < N  @*/ }

    doBarrier(){
        await (j == 0 && i < N);
        i = i + 1;
        await (i == N);
        j = j + 1;
        if (j == N){ i = 0; j = 0; }
    }
}
```

La misma contiene dos variables. La variable i cuenta los procesos esperando a completar el cupo de N para llevar a cabo la acción. La variable j cuenta la cantidad de procesos que pasaron la barrera. Las dos esperas en las operaciones **await** bloquean los procesos. La primera condición $j = 0 \wedge i < N$ deja pasar los procesos mientras la cantidad de ellos sea menor a N y hayan finalizado la ejecución de la barrera N procesos (al final de la barrera el último asigna 0 al contador j). La segunda condición $i = N$ bloquea los procesos hasta que el cupo sea N. En el caso que hayan pasado N procesos por esta espera, el último asigna 0 a las variables i y j, permitiendo desbloquear N procesos en la primera espera.

La implementación de este monitor obtenida por nuestro método es:

```
Monitor barrier {
  int i, j, N; cond c1, c2;

    doBarrier(){
      while (!( j == 0 && i < N ))
          wait(c1);
      i = i + 1;
      if (!i == N){
          signal(c1);
          do wait(c2);
          while (!i == N);
      }
      j = j + 1;
      if (j == N) { i = 0; j = 0; }
      signal(c1);
      signal(c2);
   }
}
```

Según este resultado, se puede eliminar solo la señalización dentro de la implementación de la primera operación **await**. Las demás fueron probadas necesarias en el cálculo del punto fijo.

Para obtener esta solución el prototipo necesita un poco más de 2 minutos. La cantidad de transiciones comienza en 96 quedando solo 69 después de la eliminación de las mismas en la etapa inicia. Este es el primer ejemplo donde se eliminan transiciones en la etapa inicial (los detalles de todos los ejemplos pueden verse en la tabla 5.1, pág. 200).

5.8.7 Productor/Consumidor goloso M

Este problema es el presentado en la sección 4.8.6 del capítulo anterior. En aquella ocasión el sistema no pudo detectar la eliminación de la guarda donde un consumidor libera a otro proceso de su mismo tipo.

El monitor con señalamiento automático que resuelve el problema es:

```
Monitor bbuffer {
int n, N, M;

    bbuffer(){ /*@ require: M <= N && n == 0 && M > 0 @*/ }

    producer(){
        await (n<N);
        n = n + 1;
    }
    consumer(){
        await (n>=M);
        n = n - M;
    }
}
```

En este caso nuestro método encuentra la solución correcta:

```
Monitor bbuffer {
  int n, N, M; cond c1, c2;

    producer(){
      while (!n < N)
          wait(c1);
      n = n + 1;
      signal(c1);
      signal(c2);
    }

    consumer(){
      while (!M <= n)
          wait(c2);
      n = n - M;
      signal(c1);
    }
}
```

Como puede observarse, la señalización al finalizar un consumidor hacia otro proceso de su mismo tipo es eliminada. Además se eliminan las señalizaciones en las implementaciones de las operaciones **await**. Todas las señalizaciones restantes fueron probadas como necesarias a partir del cálculo de punto fijo.

El proceso para encontrar esta solución se lleva a cabo en 6 minutos. Comparando con los 26 segundo que se necesitan para encontrar la solución al problema de productores consumidores clásico (sección 5.8.2) puede percibirse la mayor complejidad que posee el problema actual.

5.8.8 Rendezvous

Un problema clásico en concurrencia es el denominado *rendezvous*. El mismo es presentado como un mecanismo de comunicación sincrónica entre un cliente y un servidor. En esta comunicación el cliente envía un dato al servidor, el cual es procesado por este último. Durante el tiempo de procesamiento el cliente debe permanecer bloqueado hasta que el servidor termine su tarea y envíe el resultado al cliente. A su vez, el servidor procesa solo un dato al mismo tiempo.

Como puede notarse, el problema es similar al de productores y consumidores con buffer de tamaño uno, con la diferencia que los procesos deben sincronizarse mutuamente al momento del procesamiento del dato.

Una implementación posible con monitores de señalamiento automático es elaborando dos procedimientos, uno para ser invocado por el servidor y otro por los clientes:

```
Monitor client_server {
int s, d;

    client_server(){ /*@ require: s == 0 && d == 0 @*/ }

    server(){
      await (s == 0 && d == 0);
      s = s + 1;
      await (d == 1);
      s = s - 1;
    }

    client(){
      await (s == 1 && d == 0);
      d = d + 1;
      await (s == 0);
      d = d - 1;
    }
}
```

La variable `s` indica que el servidor brinda su servicio a los cliente: si es igual a 1 el servidor es accesible y 0 en caso contrario. La variable `d` indica el envío de un dato al servidor, por parte de un cliente: si es igual a 1 el dato ha sido enviado y se esta esperando la respuesta del servidor, y es igual a 0 cuando finaliza esta espera.

Ambos procedimientos contienen dos operaciones bloqueantes **await**. En el servidor, primero se espera a que algún servicio previo haya finalizado (`s==0`) y

que el resultado del procesamiento haya sido recibido (`d==0`). Una vez terminada la espera, se coloca al servidor en estado accesible y se espera a que un dato haya sido enviado por parte de un cliente (`d==1`). En el cliente, la primera espera tiene como condición que el servidor esté accesible (`s==1`) y que no haya sido enviado un dato por otro cliente (`d==0`). Una vez terminada esta espera, se envía el dato (incrementando `d`) y se espera a que el servidor termine el procesamiento (`s==0`).

Notar, que por la cantidad de operaciones **await** dentro de los procedimientos, este problema resulta difícil de tratar: estas operaciones dispuestas secuencialmente en un mismo procedimiento hace que la cantidad de transiciones se multiplique por las bifurcaciones condicional que aparecen en sus implementaciones. En la etapa inicial de generación de transiciones realizada por nuestro prototipo la cantidad de las mismas es de 740, siendo mayor en orden de magnitud a las consideradas en los ejemplos anteriores.

El prototipo devuelve la siguiente solución:

```
Monitor client_server {
  int s, d; cond c1, c2, c3, c4;

    server(){
      while (!( s == 0 && d == 0 ))
          wait(c1);
      s = s + 1;
      if (!d == 1){
          signal(c3);
          do
            wait(c2);
          while (!d == 1);}
      s = s - 1;
      signal(c4);
    }

    client(){
      while (!( s == 1 && d == 0 ))
          wait(c3);
      d = d + 1;
      if (!s == 0){
          signal(c2);
          do
            wait(c4);
          while (!s == 0);}
      d = d - 1;
      signal(c1);
    }
}
```

La solución muestra el comportamiento operacional clásico del rendezvous como una alternancia entre la ejecución del servidor y el cliente: un servidor comienza su ejecución si se satisface su condición inicial. En este caso, incrementa la variable `s` y señaliza algún cliente en espera. Un cliente, ya sea esperando en la momento de la señalización o invocando su procedimiento después de la misma, incrementa la variable `d` (enviando un dato), señaliza al servidor, y espera

por el procesamiento del dato. El servidor señalizado decrementa la variable `s` (procesa el dato) y señaliza al consumidor. Este último, recomienza su ejecución decrementando la variable `d` y señalizando a otro posible servidor bloqueado.

En total hubo 20 señalizaciones a procesar, de las cuales 16 fueron eliminadas y las restantes, que aparecen en el programa, fueron probadas necesarias. El tiempo de procesamiento total fue de 25 minutos. La cantidad de transiciones disminuyó a lo largo de todo el proceso. En la etapa inicial se eliminaron transiciones utilizando el invariante lineal, pasando de 740 transiciones a solo 217. En la segunda etapa, a medida que se fueron eliminando las señalizaciones, se llegó a solo 69 transiciones a procesar.

5.8.9 Resumen

Finalmente haremos un resumen de los resultados anteriores, agregando algunos casos de prueba que son variaciones de los anteriores problemas. En la tabla 5.1 (pág. 200) se muestran algunos valores obtenidos del procesamiento de los problemas. La columnas indican el problema, la cantidad de señales a procesar, la cantidad de transiciones iniciales, la cantidad de transiciones después de la eliminación en la primera etapa, la cantidad final y el tiempo de cómputo, en este orden. En cada fila se muestran los datos de cada problema, los cuales son (en este orden):

Sem: Problema de implementación de semáforos generales en sección 5.8.1.

P/C: Problema de productores y consumidores en sección 5.8.2.

SincF: Problema de sincronización de fase en sección 5.8.3.

L/E1: Problema de lectores y escritores en sección 5.8.4.

L/E2: Mismo problema anterior pero sin las precondiciones a la salida, como se explica en aquella sección.

Baño1: Problema del baño unisex en la sección 5.8.5.

Baño2: Variación del problema anterior donde tanto mujeres como hombres acceden si no hay personas del otro sexo y sin restricción de cantidad. Esto es, la condición para la entrada de hombres es $w = 0$ (no hay mujeres) y para mujeres es $m = 0$ (no hay hombres).

Baño3: Variación del problema original donde hombres y mujeres se comportan de forma simétrica: tanto hombres como mujeres ingresan al baño si no hay personas de sexo opuesto y la cantidad no excede cierto valor. Esto es, solo la condición para el ingreso de hombres cambia por $m < N \wedge w = 0$.

Barrera1: Problema de la sección 5.8.6.

Barrera2: Variación del problema anterior donde se aprovecha la implementación particular realizada para las operaciones **await**. En (5.7) (pág. 162) puede notarse que si la condición de la operación se cumple, el programa sigue su hilo de ejecución. Eso hace que pueda simplificarse la condición de la primera operación **await**.$(j = 0 \wedge i < N)$ por **await**.$(j = 0)$ ya que en la segunda de estas operaciones, el último proceso (el N-ésimo, cuando $i = N$) no se bloquea y cambia valor de j, bloqueando otros procesos

que intenten invocar el método (en el primer **await**). Aplicando solo esta simplificación generamos la variación del problema original.

P/CG: Problema Productor/Consumidor donde el consumidor elabora de a dos elementos. El problema fue planteado en el capítulo anterior sección 4.8.5 (pág. 145).

P/CGM: Problema de consumidor goloso en la sección 5.8.7.

Rendez: Problema de rendezvous en sección 5.8.8.

Problema	#señal	elim.	#trs. inic.	#trs 1E	#trs. fin.	Wall T.
Sem	2	1	16	16	12	2.782”
P/C	6	4	66	66	48	24.414”
SincF	6	4	66	66	48	22.734”
L/E1	10	6	90	90	48	54.057”
L/E2	10	5	90	90	60	47.782”
Baño1	10	5	90	90	54	67.042”
Baño2	10	6	90	90	54	63.630”
Baño3	10	4	90	90	60	148.333”
Barrera1	4	1	96	72	69	139.400”
Barrera2	4	1	96	72	69	145.246”
P/CG	6	3	66	66	60	127.701”
P/CGM	6	3	66	66	60	356.407”
Rendez	20	16	740	217	69	1546.356”

Tabla 5.1: Resumen de resultados

En todos los problemas se detectaron tanto las señalizaciones superfluas como las necesarias.

5.9 Conclusiones y trabajos futuros

El objetivo del trabajo presentado en este capítulo es el desarrollo de una implementación eficientes de monitores con señalamiento automático. En el capítulo anterior efectuamos una tarea similar: se implementaron regiones críticas condicionales con semáforos binarios (mediante la técnica SBD) y se desarrolló un método para mejorar su eficiencia. De manera análoga, en el capítulo actual, utilizamos monitores con señalamiento explícito para implementar monitores con señalamiento automático y se realizaron mejoras automáticas sobre estas implementaciones.

La clave de este trabajo fue la modelización del comportamiento de las implementaciones de los monitores con sistema de transiciones. En comparación con el trabajo del capítulo anterior, la cantidad de transiciones es mayor, lo cual hizo necesario emplear diversas estrategias de eliminación de transiciones.

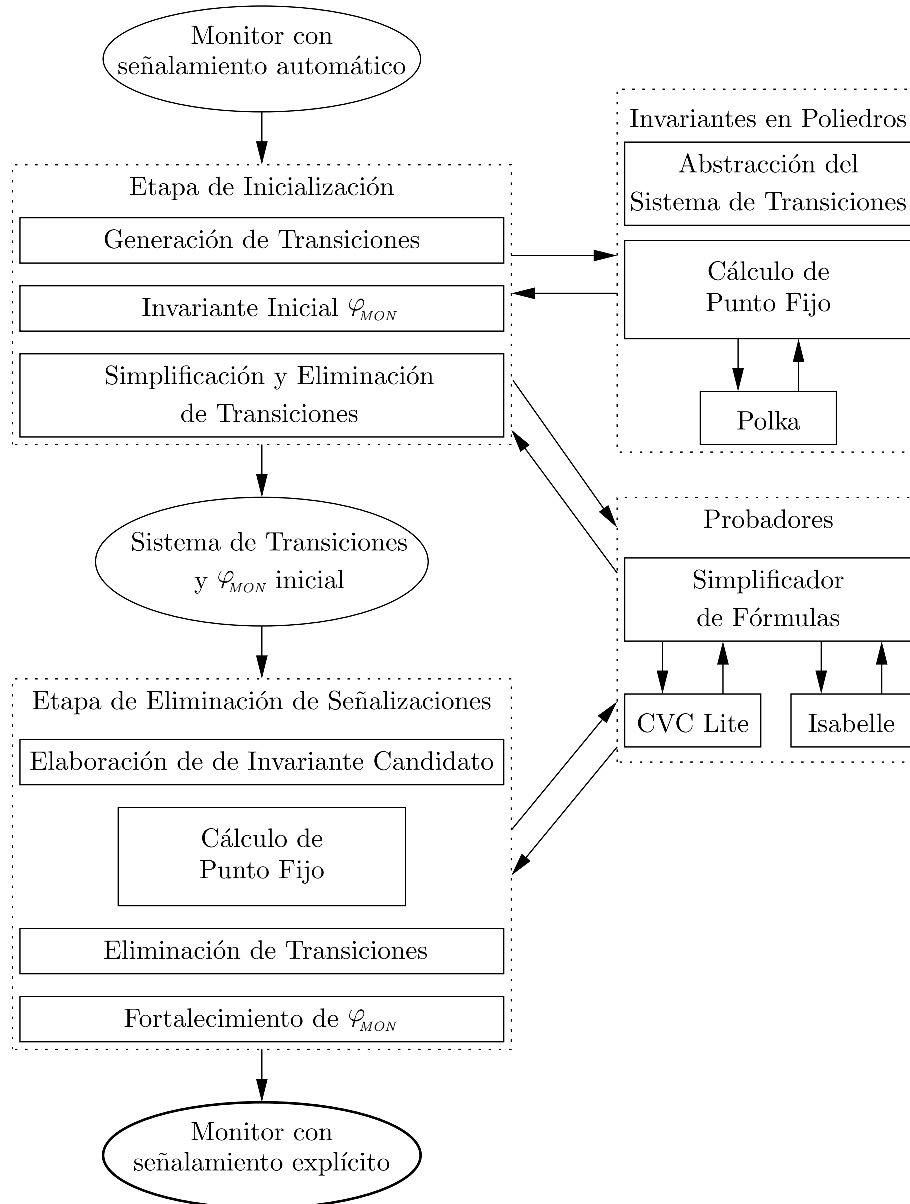
Las mismas fueron esenciales al momento de obtener los resultados mostrados en la sección anterior. En este sentido, cabe destacar que los problemas tratados no incluyen ninguna clase de invariante inicial, por lo que fue necesario incluir la técnica de interpretación abstracta en el dominio de poliedros vista en el capítulo 3. Con este invariante fue posible no solo agilizar el cálculo de punto fijo mediante simplificación de fórmulas, si no también eliminar transiciones superfluas. Al igual que en los trabajos futuros recomendados en el capítulo anterior, sería interesante comparar otras técnicas de interpretación abstracta para obtener invariantes iniciales, como por ejemplo abstracción por predicados con refinamiento por contraejemplos [CGJ+00, DD02]. Por otro lado, también se podría desarrollar alguna técnica de narrowing sobre el cálculo del máximo punto fijo al estilo de la presentada en [BW10].

Otra diferencia importante con el trabajo anterior, es que las implementaciones no cuentan de manera explícita con contadores de procesos en espera, lo cual hizo necesario incluir estas cantidades en la forma de abstracción de las colas de espera. Este tratamiento produce sistemas de transiciones de tamaño tratable y suficientemente expresivos para poder utilizarlos en el proceso automático de eliminación de señalizaciones superfluas sobre un amplio espectro de casos de prueba. Sin embargo, problemas que impliquen el uso de variables locales, como por ejemplo *Disk schedulling* [And91, pág. 295] no pueden ser abordados por la presentación actual de nuestro método. El único camino para tratar este problema sin pérdida de información es incluir la pila de variables locales de cada proceso en el sistema de transiciones. Necesariamente esto rompe con la abstracción de las colas de procesos antes mencionada. Una posible solución a esta restricción es incluir sentencias **havoc** sobre las variables locales, como se propone en [BL05] y se menciona en la sección 5.4. Con ello se produce una pérdida de información importante ya que estas variables quedan indeterminadas en las pruebas. De todas maneras queda como trabajo futuro intentar esta estrategia.

Como mencionamos en la sección 4.3 (pág. 117), inicialmente Hoare [Hoa74] presenta la idea de la técnica SBD como una forma de implementar monitores. Aunque en este artículo, por razones de eficiencia, la misma no es recomendada por su autor, más tarde E.W. Dijkstra en [Dij79] sistematiza la técnica para implementar regiones críticas condicionales, mostrando que con las simplificaciones manuales se pueden obtener programas eficientes. Queda entonces como trabajo futuro la adaptación de la técnica SBD a implementaciones de monitores con señalamiento automático, con el objetivo de aplicar la metodología desarrollada en nuestro trabajo.

Otro trabajo futuro consiste en obtener diferentes medidas de performance para comparar las soluciones con y sin las mejoras obtenidas mediante nuestro método. En [BH05] se simulan distintas implementaciones de monitores con señalamiento automático en diferentes lenguajes de programación, lo cual permite obtener este tipo de mediciones para distintos escenarios. Siguiendo esta metodología se podrían comparar estos resultados con los obtenidos sobre simulaciones que corran nuestras soluciones.

Figura 5.1: Diagrama de Diseño.



Apéndice 5.A Reglas derivación de transiciones

- $v := H \vdash \ (\bullet,\ \square\ \mathsf{true} \mapsto v := H,\ \bullet)$ *Assign*

- $\mathbf{skip} \vdash \ (\bullet,\ \square\ \mathsf{true} \mapsto \mathbf{skip},\ \bullet)$ *Skip*

- $[B] \vdash \ (\bullet,\ \square\ B \mapsto \mathbf{skip},\ \bullet)$ *Assum*

- $$\frac{S_1 \vdash \ (L,\ B_1 \mapsto P_1,\ \bullet) \quad S_2 \vdash \ (\bullet,\ B_2 \mapsto P_2,\ L')}{S_1; S_2 \vdash \ (L,\ B_1 \wedge \mathrm{wlp}.P_1.B_2 \mapsto P_1; P_2,\ L')} \ \textit{Seq1}$$

- $$\frac{S_1 \vdash \ (L,\ B \mapsto P,\ L')}{S_1; S_2 \vdash \ (L,\ B \mapsto P,\ L')} \ \textit{Seq2} \quad \text{si } L' \neq \bullet$$

- $$\frac{S_2 \vdash \ (L,\ B \mapsto P,\ L')}{S_1; S_2 \vdash \ (L,\ B \mapsto P,\ L')} \ \textit{Seq3} \quad \text{si } L \neq \bullet$$

- $$\frac{S_i \vdash \ (\bullet,\ \square\ B \mapsto P,\ L')}{\underline{\mathbf{if}}\ \langle \square j\ : 1 \leqslant j \leqslant n :\ C_j \mapsto S_j \rangle\ \underline{\mathbf{fi}}\ \vdash \ (\bullet,\ \square\ C_i \wedge B \mapsto P,\ L')} \ \textit{If1}$$

- $$\frac{S_i \vdash \ (L,\ \square\ B \mapsto P,\ L')}{\underline{\mathbf{if}}\ \langle \square j\ : 1 \leqslant j \leqslant n :\ C_j \mapsto S_j \rangle\ \underline{\mathbf{fi}}\ \vdash \ (L,\ \square\ B \mapsto P,\ L')} \ \textit{If2} \quad \text{si } L \neq \bullet$$

- $\mathbf{enter} \vdash \ (e,\ \square\ \mathsf{true} \mapsto \mathbf{skip},\ \bullet)$ *Enter*

- $\mathbf{waitLoop}_i \vdash \ (\bullet,\ \square\ \mathsf{true} \mapsto w_i := w_i + 1,\ e)$ *WakeEnter*
  con $i = 1, \cdots, m$.

- $\mathbf{waitLoop}_i \vdash \ (\bullet,\ \square\ s_j > 0 \mapsto w_i, s_j := w_i + 1, s_j - 1,\ c_j)$ *WakeWait*
  con $i = 1, \cdots, m$ y $j = 1, \cdots, m$.

- $\mathbf{waitLoop}_i \vdash \ (c_i,\ \square\ \mathsf{B}_i \mapsto \mathbf{skip},\ \bullet)$ *Wake*
  con $i = 1, \cdots, m$.

- $\mathbf{waitLoop}_i \vdash \ (c_i,\ \square\ \neg \mathsf{B}_i \mapsto w_i := w_i + 1,\ e)$ *LoopEnter*
  con $i = 1, \cdots, m$.

- $\mathbf{waitLoop}_i \vdash \ (c_i,\ \square\ \neg \mathsf{B}_i \wedge s_j > 0 \mapsto w_i, s_j := w_i + 1, s_j - 1,\ c_j)$ *LoopWait*
  con $i = 1, \cdots, m$ y $j = 1, \cdots, m$.

- $\mathbf{return} \vdash \ (\bullet,\ \mathbf{skip},\ e)$ *ReturnEnter*

- $\mathbf{return} \vdash \ (\bullet,\ \square\ s_j > 0 \mapsto s_j := s_j - 1,\ c_j)$ *ReturnWait*
  con $j = 1, \cdots, m$.

Apéndice 5.B Transiciones Productor/Consumidor

En el ejemplo 5.3 (pág. 171) se derivaron algunas transiciones del monitor Productor/Consumidor. En este apéndice se muestran todas las transiciones que conforman el sistema.

Transiciones derivadas del procedimiento productor:

$$(e,\ n < N \wedge 0 < w_1 \wedge 0 < w_2 \mapsto n, w_2, s_2, w_1, s_1 := n+1, w_2-1, s_2+1, w_1-1, s_1+1,\ e)\ ,$$

$$(e,\ n < N \wedge 0 < w_1 \wedge w_2 = 0 \mapsto n, w_1, s_1 := n+1, w_1-1, s_1+1,\ e)\ ,$$

$$(e,\ n < N \wedge w_1 = 0 \wedge 0 < w_2 \mapsto n, w_2, s_2 := n+1, w_2-1, s_2+1,\ e)\ ,$$

$$(e,\ n < N \wedge w_1 = 0 \wedge w_2 = 0 \mapsto n := n+1,\ e)\ ,$$

$$(e,\ n < N \wedge 0 < w_1 \wedge 0 < w_2 \wedge 0 \leqslant s_1 \mapsto n, w_2, s_2, w_1 := n+1, w_2-1, s_2+1, w_1-1,\ c_1)\ ,$$

$$(e,\ n < N \wedge 0 < w_1 \wedge w_2 = 0 \wedge 0 \leqslant s_1 \mapsto n, w_1 := n+1, w_1-1,\ c_1)\ ,$$

$$(e,\ n < N \wedge w_1 = 0 \wedge 0 < w_2 \wedge 0 < s_1 \mapsto n, s_1, w_2, s_2 := n+1, s_1-1, w_2-1, s_2+1,\ c_1)\ ,$$

$$(e,\ n < N \wedge w_1 = 0 \wedge w_2 = 0 \wedge 0 < s_1 \mapsto n, s_1 := n+1, s_1-1,\ c_1)\ ,$$

$$(e,\ n < N \wedge 0 < w_1 \wedge 0 < w_2 \wedge 0 \leqslant s_2 \mapsto n, w_2, w_1, s_1 := n+1, w_2-1, w_1-1, s_1+1,\ c_2)\ ,$$

$$(e,\ n < N \wedge 0 < w_1 \wedge w_2 = 0 \wedge 0 < s_2 \mapsto n, s_2, w_1, s_1 := n+1, s_2-1, w_1-1, s_1+1,\ c_2)\ ,$$

$$(e,\ n < N \wedge w_1 = 0 \wedge 0 < w_2 \wedge 0 \leqslant s_2 \mapsto n, w_2 := n+1, w_2-1,\ c_2)\ ,$$

$$(e,\ n < N \wedge w_1 = 0 \wedge w_2 = 0 \wedge 0 < s_2 \mapsto n, s_2 := n+1, s_2-1,\ c_2)\ ,$$

$$(e,\ \neg n < N \wedge 0 < w_2 \mapsto s_2, w_2, w_1 := s_2+1, w_2-1, w_1+1,\ e)\ ,$$

$$(e,\ \neg n < N \wedge w_2 = 0 \mapsto w_1 := w_1+1,\ e)\ ,$$

$$(e,\ \neg n < N \wedge 0 < w_2 \wedge 0 < s_1 \mapsto s_2, w_2, s_1, w_1 := s_2+1, w_2-1, s_1-1, w_1+1,\ c_1)\ ,$$

$$(e,\ \neg n < N \wedge w_2 = 0 \wedge 0 < s_1 \mapsto s_1, w_1 := s_1-1, w_1+1,\ c_1)\ ,$$

$$(e,\ \neg n < N \wedge 0 < w_2 \wedge 0 \leqslant s_2 \mapsto w_2, w_1 := w_2-1, w_1+1,\ c_2)\ ,$$

$$(e,\ \neg n < N \wedge w_2 = 0 \wedge 0 < s_2 \mapsto s_2, w_1 := s_2-1, w_1+1,\ c_2)\ ,$$

$$(c_1,\ n < N \wedge 0 < w_1 \wedge 0 < w_2 \mapsto$$
$$s_1, w_1, s_2, w_2, n := s_1 + 1, w_1 - 1, s_2 + 1, w_2 - 1, n + 1,\ e)\ ,$$

$$(c_1,\ n < N \wedge 0 < w_1 \wedge w_2 = 0 \mapsto s_1, w_1, n := s_1 + 1, w_1 - 1, n + 1,\ e)\ ,$$

$$(c_1,\ n < N \wedge w_1 = 0 \wedge 0 < w_2 \mapsto s_2, w_2, n := s_2 + 1, w_2 - 1, n + 1,\ e)\ ,$$

$$(c_1,\ n < N \wedge w_1 = 0 \wedge w_2 = 0 \mapsto n := n + 1,\ e)\ ,$$

$$(c_1,\ n < N \wedge 0 < w_1 \wedge 0 < w_2 \wedge 0 \leqslant s_1 \mapsto$$
$$w_1, s_2, w_2, n := w_1 - 1, s_2 + 1, w_2 - 1, n + 1,\ c_1)\ ,$$

$$(c_1,\ n < N \wedge 0 < w_1 \wedge w_2 = 0 \wedge 0 \leqslant s_1 \mapsto w_1, n := w_1 - 1, n + 1,\ c_1)\ ,$$

$$(c_1,\ n < N \wedge w_1 = 0 \wedge 0 < w_2 \wedge 0 < s_1 \mapsto$$
$$s_2, w_2, s_1, n := s_2 + 1, w_2 - 1, s_1 - 1, n + 1,\ c_1)\ ,$$

$$(c_1,\ n < N \wedge w_1 = 0 \wedge w_2 = 0 \wedge 0 < s_1 \mapsto s_1, n := s_1 - 1, n + 1,\ c_1)\ ,$$

$$(c_1,\ n < N \wedge 0 < w_1 \wedge 0 < w_2 \wedge 0 \leqslant s_2 \mapsto$$
$$s_1, w_1, w_2, n := s_1 + 1, w_1 - 1, w_2 - 1, n + 1,\ c_2)\ ,$$

$$(c_1,\ n < N \wedge 0 < w_1 \wedge w_2 = 0 \wedge 0 < s_2 \mapsto$$
$$s_1, w_1, s_2, n := s_1 + 1, w_1 - 1, s_2 - 1, n + 1,\ c_2)\ ,$$

$$(c_1,\ n < N \wedge w_1 = 0 \wedge 0 < w_2 \wedge 0 \leqslant s_2 \mapsto w_2, n := w_2 - 1, n + 1,\ c_2)\ ,$$

$$(c_1,\ n < N \wedge w_1 = 0 \wedge w_2 = 0 \wedge 0 < s_2 \mapsto s_2, n := s_2 - 1, n + 1,\ c_2)\ ,$$

$$(c_1,\ \neg n < N \mapsto w_1 := w_1 + 1,\ e)\ ,$$

$$(c_1,\ \neg n < N \wedge 0 < s_1 \mapsto s_1, w_1 := s_1 - 1, w_1 + 1,\ c_1)\ ,$$

$$(c_1,\ \neg n < N \wedge 0 < s_2 \mapsto s_2, w_1 := s_2 - 1, w_1 + 1,\ c_2)$$

Transiciones derivadas del procedimiento consumidor:

$$(e,\ 0 < n \wedge 0 < w_1 \wedge 0 < w_2 \mapsto$$
$$n, w_2, s_2, w_1, s_1 := n - 1, w_2 - 1, s_2 + 1, w_1 - 1, s_1 + 1,\ e)\ ,$$

$$(e,\ 0 < n \wedge 0 < w_1 \wedge w_2 = 0 \mapsto n, w_1, s_1 := n - 1, w_1 - 1, s_1 + 1,\ e)\ ,$$

$$(e,\ 0 < n \wedge w_1 = 0 \wedge 0 < w_2 \mapsto n, w_2, s_2 := n - 1, w_2 - 1, s_2 + 1,\ e)\ ,$$

$$(e,\ 0 < n \wedge w_1 = 0 \wedge w_2 = 0 \mapsto n := n - 1,\ e)\ ,$$

$$(e,\ 0 < n \wedge 0 < w_1 \wedge 0 < w_2 \wedge 0 \leqslant s_1 \mapsto$$
$$n, w_2, s_2, w_1 := n - 1, w_2 - 1, s_2 + 1, w_1 - 1,\ c_1)\ ,$$

$$(e,\ 0 < n \wedge 0 < w_1 \wedge w_2 = 0 \wedge 0 \leqslant s_1 \mapsto n, w_1 := n - 1, w_1 - 1,\ c_1)\ ,$$

$(e,\ 0 < n \wedge w_1 = 0 \wedge 0 < w_2 \wedge 0 < s_1 \mapsto$
$\quad n, s_1, w_2, s_2 := n-1, s_1-1, w_2-1, s_2+1,\ c_1)$,

$(e,\ 0 < n \wedge w_1 = 0 \wedge w_2 = 0 \wedge 0 < s_1 \mapsto n, s_1 := n-1, s_1-1,\ c_1)$,

$(e,\ 0 < n \wedge 0 < w_1 \wedge 0 < w_2 \wedge 0 \leqslant s_2 \mapsto$
$\quad n, w_2, w_1, s_1 := n-1, w_2-1, w_1-1, s_1+1,\ c_2)$,

$(e,\ 0 < n \wedge 0 < w_1 \wedge w_2 = 0 \wedge 0 < s_2 \mapsto$
$\quad n, s_2, w_1, s_1 := n-1, s_2-1, w_1-1, s_1+1,\ c_2)$,

$(e,\ 0 < n \wedge w_1 = 0 \wedge 0 < w_2 \wedge 0 \leqslant s_2 \mapsto n, w_2 := n-1, w_2-1,\ c_2)$,

$(e,\ 0 < n \wedge w_1 = 0 \wedge w_2 = 0 \wedge 0 < s_2 \mapsto n, s_2 := n-1, s_2-1,\ c_2)$,

$(e,\ \neg 0 < n \wedge 0 < w_1 \mapsto s_1, w_1, w_2 := s_1+1, w_1-1, w_2+1,\ e)$,

$(e,\ \neg 0 < n \wedge w_1 = 0 \mapsto w_2 := w_2+1,\ e)$,

$(e,\ \neg 0 < n \wedge 0 < w_1 \wedge 0 \leqslant s_1 \mapsto w_1, w_2 := w_1-1, w_2+1,\ c_1)$,

$(e,\ \neg 0 < n \wedge w_1 = 0 \wedge 0 < s_1 \mapsto s_1, w_2 := s_1-1, w_2+1,\ c_1)$,

$(e,\ \neg 0 < n \wedge 0 < w_1 \wedge 0 < s_2 \mapsto$
$\quad s_1, w_1, s_2, w_2 := s_1+1, w_1-1, s_2-1, w_2+1,\ c_2)$,

$(e,\ \neg 0 < n \wedge w_1 = 0 \wedge 0 < s_2 \mapsto s_2, w_2 := s_2-1, w_2+1,\ c_2)$,

$(c_2,\ 0 < n \wedge 0 < w_1 \wedge 0 < w_2 \mapsto$
$\quad s_1, w_1, s_2, w_2, n := s_1+1, w_1-1, s_2+1, w_2-1, n-1,\ e)$,

$(c_2,\ 0 < n \wedge 0 < w_1 \wedge w_2 = 0 \mapsto s_1, w_1, n := s_1+1, w_1-1, n-1,\ e)$,

$(c_2,\ 0 < n \wedge w_1 = 0 \wedge 0 < w_2 \mapsto s_2, w_2, n := s_2+1, w_2-1, n-1,\ e)$,

$(c_2,\ 0 < n \wedge w_1 = 0 \wedge w_2 = 0 \mapsto n := n-1,\ e)$,

$(c_2,\ 0 < n \wedge 0 < w_1 \wedge 0 < w_2 \wedge 0 \leqslant s_1 \mapsto$
$\quad w_1, s_2, w_2, n := w_1-1, s_2+1, w_2-1, n-1,\ c_1)$,

$(c_2,\ 0 < n \wedge 0 < w_1 \wedge w_2 = 0 \wedge 0 \leqslant s_1 \mapsto w_1, n := w_1-1, n-1,\ c_1)$,

$(c_2,\ 0 < n \wedge w_1 = 0 \wedge 0 < w_2 \wedge 0 < s_1 \mapsto$
$\quad s_2, w_2, s_1, n := s_2+1, w_2-1, s_1-1, n-1,\ c_1)$,

$(c_2,\ 0 < n \wedge w_1 = 0 \wedge w_2 = 0 \wedge 0 < s_1 \mapsto s_1, n := s_1-1, n-1,\ c_1)$,

$(c_2,\ 0 < n \wedge 0 < w_1 \wedge 0 < w_2 \wedge 0 \leqslant s_2 \mapsto$
$\quad s_1, w_1, w_2, n := s_1+1, w_1-1, w_2-1, n-1,\ c_2)$,

$$(c_2,\ 0 < n \wedge 0 < w_1 \wedge w_2 = 0 \wedge 0 < s_2 \mapsto s_1, w_1, s_2, n := s_1 + 1, w_1 - 1, s_2 - 1, n - 1,\ c_2)\ ,$$

$$(c_2,\ 0 < n \wedge w_1 = 0 \wedge 0 < w_2 \wedge 0 \leqslant s_2 \mapsto w_2, n := w_2 - 1, n - 1,\ c_2)\ ,$$

$$(c_2,\ 0 < n \wedge w_1 = 0 \wedge w_2 = 0 \wedge 0 < s_2 \mapsto s_2, n := s_2 - 1, n - 1,\ c_2)\ ,$$

$$(c_2,\ \neg 0 < n \mapsto w_2 := w_2 + 1,\ e)\ ,$$

$$(c_2,\ \neg 0 < n \wedge 0 < s_1 \mapsto s_1, w_2 := s_1 - 1, w_2 + 1,\ c_1)\ ,$$

$$(c_2,\ \neg 0 < n \wedge 0 < s_2 \mapsto s_2, w_2 := s_2 - 1, w_2 + 1,\ c_2)$$

Apéndice 5.C Demostración Teorema 5.4

Primero demostraremos que dado el cuerpo de un procedimiento en un monitor con señalamiento automático, que no incluya las primitivas **enter** al comienzo y **return** al final, si se verifican ciertas pre y poscondiciones junto con la condición de prueba sobre **await**, entonces las mismas se preservan en las transiciones generadas desde su implementación.

Lema 5.7
Sea $\tilde{S}$ el programa resultado de aplicar la implementación de las operaciones **await** en (5.11) (pág. 166) sobre un programa S que contiene solo sentencias de asignación, **if**, **skip**, y sentencias de espera **await**.B_i con $i = 1, \cdots, m$ (no contiene las primitivas **enter** ni **return**). Además, el programa S verifica

$$\{p\}\ S\ \{q\}$$

a partir de las condiciones de prueba

$$\{I\}\ \textbf{await}.\mathsf{B}_i\ \{I \wedge \mathsf{B}_i\} \quad \text{con} \quad i = 1, \cdots, m$$

donde en p, q e I no aparecen las variables de longitud de las colas (w_i y s_i). Entonces para toda transición derivada desde $\tilde{S}$

$$\tilde{S} \vdash (l, G, l')$$

a partir del conjunto de reglas en la sección 5.4, se cumple alguna de las siguientes proposiciones:

1. si $l = \bullet$ y $l' = \bullet$ entonces $\{p\}\ G\ \{q\}$,
2. si $l \in \{e, c_1, \cdots, c_m\}$ y $l' = \bullet$ entonces $\{I\}\ G\ \{q\}$,
3. si $l = \bullet$ y $l' \in \{e, c_1, \cdots, c_m\}$ entonces $\{p\}\ G\ \{I\}$,
4. si $l, l' \in \{e, c_1, \cdots, c_m\}$ entonces $\{I\}\ G\ \{I\}$.

DEMOSTRACIÓN
Probaremos el lema por inducción estructural en la gramática del programa S comenzando por los casos bases. Primero analizaremos el caso donde el programa consiste solo de una sentencia simple.

Caso sentencias simples (asignación o **skip**) Si $S \doteq v := H$ por hipótesis del lema se cumple:

$$\{p\}\ v := H\ \{q\} \ .$$

Además, la implementación $\tilde{S}$ será el mismo programa S. Demostraremos entonces que las transiciones generadas por esta sentencia cumplen las proposiciones del lema. Para este caso en particular se generará únicamente la transición $(\bullet, \square\ \mathsf{true} \mapsto v := H, \bullet)$ ya que solo se puede aplicar la regla *Assign* (pág. 165) a esta sentencia. Por lo tanto, como ambas locaciones son la auxiliar tendremos que demostrar solo $\{p\}\ \square\ \mathsf{true} \mapsto v := H\ \{q\}$ (primera proposición). Esto es verdadero a partir de la hipótesis $\{p\}\ v := H\ \{q\}$.
La demostración del **skip** es similar.

A continuación demostraremos el caso en que el programa consiste de una operación **await**.

Caso $S \doteq$ **await**B_i Por hipótesis del teorema el programa S verifica las precondición p y poscondición q a partir de la condición de prueba

$$\{I\}\ \textbf{await}.\mathsf{B}_i\ \{I \wedge \mathsf{B}_i\}$$

Por lo tanto $p \Rightarrow I$ y $I \wedge \mathsf{B}_i \Rightarrow q$.

Además, el programa $\tilde{S}$ será la implementación de esta operación dada en (5.11) (pág. 166):

$$
\begin{array}{lll}
\textbf{await}.\mathsf{B}_i & \doteq & \underline{\textbf{if}}\ \ \mathsf{B}_i \mapsto \textbf{skip} \\
 & & \square\ \neg\mathsf{B}_i \mapsto \\
 & & \quad \textbf{for}\ (j = 1, \cdots, m \wedge j \neq i) \\
 & & \quad\quad \underline{\textbf{if}}\ w_j > 0 \mapsto w_j, s_j := w_j - 1, s_j + 1 \\
 & & \quad\quad \square\ \ w_j = 0 \mapsto \textbf{skip} \\
 & & \quad\quad \underline{\textbf{fi}}; \\
 & & \quad \textbf{waitLoop}_i \\
 & & \underline{\textbf{fi}}
\end{array}
$$

Demostraremos entonces que todas las transiciones generadas desde esta implementación verifican las cuatro propiedades del lema. La única transición que posee ambas locaciones auxiliares es la que se deriva de la primer rama del **if** externo (con condición B_i):

$$(\bullet,\ \square\ \mathsf{B}_i \mapsto \textbf{skip}, \bullet)$$

la cual es derivada usando las reglas *If1* y *Skip*. Calculando la weakest liberal precondition sobre esta sentencia, obtenemos:

$$\text{wlp}.(\square\ \mathsf{B}_i \mapsto \textbf{skip}).(I \wedge \mathsf{B}_i) = \neg\mathsf{B}_i\ \vee\ I$$

con lo que se verifica $\{I\}\ \square\ \mathsf{B}_i \mapsto \textbf{skip}\ \{I \wedge \mathsf{B}_i\}$. Por lo tanto, utilizando las hipótesis iniciales $p \Rightarrow I$ y $I \wedge \mathsf{B}_i \Rightarrow q$, se verifica la primer proposición:

$$\{p\}\ \square\ \mathsf{B}_i \mapsto \textbf{skip}\ \{q\}\ \ .$$

A continuación demostraremos que las proposiciones del lema se cumplen para las transiciones transiciones restantes generadas en la segunda rama del **if** externo. Dentro de la misma aparece un bucle **for** compuesto secuencialmente con el programa $\textbf{waitLoop}_i$. Como ya vimos, aquel bucle es equivalente a $m - 1$ sentencias **if** compuestas secuencialmente. Cada uno de ellas produce las transiciones

$$
\begin{array}{l}
(\bullet,\ \square\ w_j > 0 \mapsto w_j, s_j := w_j - 1, s_j + 1,\ \bullet) \\
(\bullet,\ \square\ w_j = 0 \mapsto \textbf{skip},\ \bullet)
\end{array}
$$

con $1 \leqslant j \leqslant m \wedge j \neq i$. Para generar las transiciones finales, cada una se concatena con las transiciones generadas por las subsiguientes sentencias **if** aplicando la regla *Seq1*, obteniéndose así 2^{m-1} transiciones. Las mismas se pueden construir de la siguiente manera: sea el conjunto de índices

$J \doteq \{j \mid 1 \leqslant j \leqslant m \wedge j \neq i\}$, entonces para cada subconjunto $K \subseteq J$ tendremos una transición

$$(\bullet,\ \square\ \langle \forall\, k\ : k \in K :\ w_k > 0 \,\rangle \wedge \langle \forall\, k\ : k \in J - K :\ w_k = 0 \,\rangle \mapsto \\ \langle (;)\, k\ : k \in K :\ w_k, s_k := w_k - 1, s_k + 1 \,\rangle,\ \bullet)$$

la cual modela las ejecuciones donde, si $k \in K$, la condición $w_k > 0$ de la k-ésima sentencia **if** es satisfecha, y en el caso que k no pertenezca al conjunto, la condición $w_k = 0$ lo será. De esta manera, la guarda de la transición indica las condiciones satisfechas ($w_k > 0$ o $w_k = 0$) en la ejecución de la composición secuencial de las sentencias **if** y la asignación es la resultante de las ramas ejecutadas en estas sentencias.

A su vez, estas 2^{m-1} transiciones se concatenan con las generadas en el subsiguiente programa $\mathbf{waitLoop}_i$ por las reglas *WakeEnter* en (5.12) (pág. 168) y *WakeWait* en (5.13), obteniéndose las siguientes $(m+1) * 2^{m-1}$ transiciones:

$$(\bullet,\ \square\ \langle \forall\, k\ : k \in K :\ w_k > 0 \,\rangle \wedge \langle \forall\, k\ : k \in J - K :\ w_k = 0 \,\rangle \mapsto \\ w_i := w_i + 1; \langle (;)\, k\ : k \in K :\ w_k, s_k := w_k - 1, s_k + 1 \,\rangle,\ e)\ ,$$

$$(\bullet,\ \square\ s_j > 0 \wedge \langle \forall\, k\ : k \in K :\ w_k > 0 \,\rangle \wedge \langle \forall\, k\ : k \in K - K :\ w_k = 0 \,\rangle \mapsto \\ w_i := w_i + 1; s_j := s_j - 1; \langle (;)\, k\ : k \in K :\ w_k, s_k := w_k - 1, s_k + 1 \,\rangle,\ c_j)$$

para cada $K \subseteq J$ y $j = 1, \cdots, m$. Las transiciones son generadas dentro de la segunda rama del **if** externo (con condición $\neg \mathsf{B}_i$), por lo cual, aplicando la regla *If1* obtenemos las siguientes transiciones finales desde la implementación de la operación **await**.B_i:

$$(\bullet,\ \square \neg \mathsf{B}_i \wedge \langle \forall\, k\ : k \in K :\ w_k > 0 \,\rangle \wedge \langle \forall\, k\ : k \in J - K :\ w_k = 0 \,\rangle \mapsto \\ w_i := w_i + 1; \langle (;)\, k\ : k \in K :\ w_k, s_k := w_k - 1, s_k + 1 \,\rangle,\ e)\ ,$$

$$(\bullet,\ \square \neg \mathsf{B}_i \wedge s_j > 0 \wedge \langle \forall\, k\ : k \in K :\ w_k > 0 \,\rangle \wedge \langle \forall\, k\ : k \in K - K :\ w_k = 0 \,\rangle \mapsto \\ w_i := w_i + 1; s_j := s_j - 1; \langle (;)\, k\ : k \in K :\ w_k, s_k := w_k - 1, s_k + 1 \,\rangle,\ c_j)$$

con $K \subseteq J$ y $j = 1, \cdots, m$.
Las transiciones tienen locación de salida la auxiliar y de llegada una en $\{e, c_1, \cdots, c_m\}$, por lo cual demostraremos la tercer proposición del lema.
Notar que las transiciones solo modifican la longitud de las colas (s_i w_i con $i = 1, \cdots, m$). Por lo tanto, si $(\bullet, G, l')$ es una de estas transiciones se puede demostrar que $I \vee \mathsf{B}_i \Rightarrow \text{wlp}.G.I$ ya que I no depende de estas variables. En consecuencia, como $p \Rightarrow I$ entonces se verifica $\{p\}\ G\ \{I\}$ demostrándose así la tercer proposición.

Queda por demostrar que las restantes transiciones generadas por la implementación de **await**.B_i cumplen con las proposiciones. Las mismas son simplemente generadas por las reglas *Wake* (pág. 169), *LoopEnter* y *LoopWait*. La primera genera la transición

$$(c_i,\ \square\ \mathsf{B}_i \mapsto \mathbf{skip},\ \bullet)\ .$$

La precondición de esta sentencia con respecto a $I \wedge \mathsf{B}_i$ es

$$\text{wlp}.(\square\ \mathsf{B}_i \mapsto \mathbf{skip}).(I \wedge \mathsf{B}_i) = \neg \mathsf{B}_i\ \vee\ I\ .$$

Como $I \Rightarrow \neg \mathsf{B}_i \ \vee \ I$ e $I \wedge \mathsf{B}_i \Rightarrow q$ claramente se verifica

$$\{I\} \ \Box \ \mathsf{B}_i \mapsto \textbf{skip} \ \{q\}$$

por lo tanto se cumple la segunda proposición.

Las reglas *LoopEnter* y *LoopWait* generan las transiciones

$$\begin{array}{l}(c_i, \ \Box \neg \mathsf{B}_i \mapsto w_i := w_i + 1, \ e) \\ (c_i, \ \Box \neg \mathsf{B}_i \wedge s_j > 0 \mapsto w_i, s_j := w_i + 1, s_j - 1, \ c_j)\end{array}$$

con $j = 1, \cdots, m$. Las mismas cumplen el invariante I como pre y poscondición ya que I no depende la longitud de las colas. Por los tanto verifican la cuarta proposición.

El siguiente caso en la gramática consiste en la composición secuencial de dos subprogramas. Este es el primer caso inductivo a demostrar.

Paso inductivo $S \doteq S_1; S_2$ Como hipótesis de lema tenemos que se verifica

$$\{p\} \ S_1; S_2 \ \{q\} \ .$$

Por la regla de composición secuencial en la lógica de Hoare [AO97, cap. 3] existe un predicado intermedio r tal que

$$\{p\} \ S_1 \ \{r\} \quad \text{y} \quad \{r\} \ S_2 \ \{q\} \ .$$

A partir de este resultado, por hipótesis inductiva podemos suponer que las transiciones generadas sobre $\tilde{S}_1$ y $\tilde{S}_2$ cumplen con las proposiciones del lema. Utilizando estas hipótesis demostraremos que todas las transiciones generadas desde el programa $\tilde{S}_1; \tilde{S}_2$ también las verifican. Para ello analizaremos distintos subcasos según sus locaciones de salida y llegada.

En el caso que se genere una transición con ambas locaciones auxiliares (primer proposición del lema) de la forma:

$$\tilde{S}_1; \tilde{S}_2 \vdash (\bullet, G, \bullet)$$

la única regla que se puede haber aplicado es *Seq1* (*Seq2* y *Seq3* necesitan que alguna locación sea distinta a la auxiliar). Sean

$$\tilde{S}_1 \vdash (\bullet, \ \Box \ B_1 \mapsto P_1, \ \bullet) \quad \text{y} \quad \tilde{S}_2 \vdash (\bullet, \ \Box \ B_2 \mapsto P_2, \ \bullet)$$

las premisas para su derivación. Entonces la sentencia G queda definida por esta regla como:

$$G \ \doteq \ \Box \ B_1 \wedge \text{wlp}.P_1.B_2 \mapsto P_1; P_2$$

Por hipótesis inductiva, ambas sentencias en la premisa cumplen:

$$\{p\} \ \Box \ B_1 \mapsto P_1 \ \{r\} \quad \text{y} \quad \{r\} \ \Box \ B_2 \mapsto P_2 \ \{q\}$$

lo cual es equivalente a

$$p \wedge B_1 \Rightarrow \text{wlp}.P_1.r \quad \text{y} \quad r \wedge B_2 \Rightarrow \text{wlp}.P_2.q$$

A partir de estas hipótesis debemos demostrar la primer proposición sobre la sentencia G:

$$\{p\} \square B_1 \wedge \text{wlp}.P_1.B_2 \mapsto P_1; P_2 \ \{q\}$$

o su equivalente

$$p \Rightarrow (B_1 \wedge \text{wlp}.P_1.B_2 \Rightarrow \text{wlp}.P_1; P_2.q)$$

Tomando el termino que contiene la weakest precondition de la composición entre P_1 y $P2$ podemos demostrar que:

$$\begin{array}{cl}
 & \text{wlp}.P_1; P_2.q \\
\equiv & \{ \text{ wlp de la composición secuencial } \} \\
 & \text{wlp}.P_1.(\text{wlp}.P_2.q) \\
\Leftarrow & \{ \text{ Monotonía de wlp e hip. inductiva } r \wedge B_2 \Rightarrow \text{wlp}.P_2.q \ \} \\
 & \text{wlp}.P_1.(r \wedge B_2) \\
\equiv & \{ \text{ Conjuntividad de wlp } \} \\
 & \text{wlp}.P_1.r \wedge \text{wlp}.P_1.B_2 \\
\Leftarrow & \{ \text{ Hipótesis inductiva } p \wedge B_1 \Rightarrow \text{wlp}.P_1.r \ \} \\
 & p \wedge B_1 \wedge \text{wlp}.P_1.B_2
\end{array}$$

Por lo tanto

$$p \wedge B_1 \wedge \text{wlp}.P_1.B_2 \Rightarrow \text{wlp}.P_1; P_2.q$$

con lo cual se demuestra la primer proposición del lema.

El próximo subcaso es cuando la locación de llegada es la auxiliar:

$$\tilde{S}_1; \tilde{S}_2 \vdash (l, G, \bullet) \quad \text{con} \quad l \in \{e, c_1, \cdots, c_m\} \ .$$

Para obtener esta transición las únicas reglas de aplicación posibles son *Seq1* y *Seq3* (*Seq2* necesita que la locación de salida sea la auxiliar).
Analicemos el subcaso donde se aplica la regla *Seq1*. Si las premisas para la derivación de la transición son:

$$\tilde{S}_1 \vdash (l, \ \square B_1 \mapsto P_1, \ \bullet) \quad \text{y} \quad \tilde{S}_2 \vdash (\bullet, \ \square B_2 \mapsto P_2, \ \bullet)$$

entonces la sentencia G tiene la forma:

$$G \ \doteq \ \square B_1 \wedge \text{wlp}.P_1.B_2 \mapsto P_1; P_2$$

Por hipótesis inductiva se cumple

$$\{I\} \square B_1 \mapsto P_1 \ \{r\} \quad \text{y} \quad \{r\} \square B_2 \mapsto P_2 \ \{q\}$$

o su equivalente

$$I \wedge B_1 \Rightarrow \text{wlp}.P_1.r \quad \text{y} \quad r \wedge B_2 \Rightarrow \text{wlp}.P_2.q$$

Reemplazando p por I en la prueba anterior se demuestra que

$$I \Rightarrow (B_1 \wedge \text{wlp}.P_1.B_2 \Rightarrow \text{wlp}.P_1; P_2.q)$$

con lo cual se demuestra las segunda proposición del lema:

$$\{I\}\ \Box\ B_1 \wedge \text{wlp}.P_1.B_2 \mapsto P_1; P_2\ \{q\}$$

En el caso que la transición sea generada por aplicación de la regla *Seq3* con premisa

$$\tilde{S}_2 \vdash (l,\ \Box\ B \mapsto P,\ \bullet) \quad \text{con} \quad l \in \{e, c_1, \cdots, c_m\}$$

entonces la sentencia G queda definida como $G \doteq \Box\ B \mapsto P$. Por hipótesis inductiva en $\tilde{S}_2$ se verifica la segunda proposición sobre G:

$$\{I\}\ \Box\ B \mapsto P\ \{q\} \ \ .$$

El próximo subcaso es cuando la locación de salida es la auxiliar:

$$\tilde{S}_1; \tilde{S}_2 \vdash (\bullet, G, l') \quad \text{con} \quad l' \in \{e, c_1, \cdots, c_m\} \ \ .$$

Para obtener esta transición las únicas reglas de aplicación posibles son *Seq1* y *Seq2* (*Seq3* necesita que la locación de llegada sea la auxiliar). Las demostraciones para ambas reglas son simétricas a las anteriores, verificando de esta manera la tercer proposición:

$$\{p\}\ G\ \{I\} \ \ .$$

Solo queda por demostrar el subcaso donde ambas locaciones son distintas a la auxiliar:

$$\tilde{S}_1; \tilde{S}_2 \vdash (l, G, l') \quad \text{con} \quad l, l' \in \{e, c_1, \cdots, c_m\}$$

Las reglas de posible aplicación para obtener esta transición son *Seq1*, *Seq2* y *Seq3*. Para la demostración sobre la regla *Seq1* se realiza la misma prueba anterior reemplazando p y q por I. De esta manera se demuestra que se verifica la cuarta proposición. Las demostraciones sobre las dos reglas siguientes se realizan utilizando directamente las hipótesis inductivas, verificando de esta manera la cuarta proposición.

El último caso en la gramática del programa S es la sentencia condicional **if**.

Paso inductivo $S \doteq \underline{\textbf{if}}\ \langle \Box j\ : 1 \leqslant j \leqslant n :\ C_j \mapsto S_j \rangle\ \underline{\textbf{fi}}$ Por hipótesis del lema se cumple:

$$\{p\}\ \underline{\textbf{if}}\ \langle \Box j\ : 1 \leqslant j \leqslant n :\ C_j \mapsto S_j \rangle\ \underline{\textbf{fi}}\ \{q\}$$

y por la regla del comando de alternativa en la lógica de Hoare [AO97, cap. 10] se verifica:

$$\{p \wedge C_i\}\ S_i\ \{q\} \quad \text{con} \quad i = 1, \cdots, n \tag{5.24}$$

Sea la implementación de S:

$$\tilde{S} \doteq \underline{\textbf{if}}\ \langle \Box j\ : 1 \leqslant j \leqslant n :\ C_j \mapsto \tilde{S}_j \rangle\ \underline{\textbf{fi}}$$

y supongamos que a partir de esta sentencia se genera una transición con locación de salida igual a la auxiliar:

$$\underline{\textbf{if}}\ \langle \Box j\ : 1 \leqslant j \leqslant n :\ C_j \mapsto \tilde{S}_j \rangle\ \underline{\textbf{fi}} \vdash (\bullet, G, l')$$

La misma solo puede ser generada a partir de la aplicación de la regla *If1* por la restricción en la locación de salida. Sea su premisa igual a:

$$\tilde{S}_i \;\vdash\; (\bullet,\; \square\, B \mapsto P,\; l')$$

por lo tanto la sentencia G debe tener la forma:

$$G \;\doteq\; \square\, C_i \wedge B \mapsto P \;\;.$$

Analicemos dos subcasos: $l' = \bullet$ y $l' \in \{e, c_1, \cdots, c_m\}$. En el primero, por (5.24) y aplicando hipótesis inductiva se cumple

$$\{p \wedge C_i\} \,\square\, B \mapsto P \,\{q\} \;\;.$$

Por lo tanto también se puede demostrar fácilmente que

$$\{p\} \,\square\, C_i \wedge B \mapsto P \,\{q\}$$

con lo cual, por la definición anterior de G, se cumple la primer proposición del lema.

El segundo subcaso ($l' \in \{e, c_1, \cdots, c_m\}$) se demuestra de forma análoga: por hipótesis inductiva y (5.24) se cumple

$$\{p \wedge C_i\} \,\square\, B \mapsto P \,\{I\}$$

con lo cual se puede demostrar

$$\{p\} \,\square\, C_i \wedge B \mapsto P \,\{I\}$$

Por definición de G este resultado prueba la tercer proposición del lema.

Queda por demostrar el caso donde la transición generada por $\tilde{S}$ es distinta a la auxiliar:

$$\underline{\mathbf{if}}\ \langle \square j\ : 1 \leqslant j \leqslant n :\ C_j \mapsto \tilde{S}_j \,\rangle\ \underline{\mathbf{fi}} \;\vdash\; (l, G, l') \quad \text{con} \quad l \in \{e, c_1, \cdots, c_m\} \;\;.$$

Por la restricción sobre la locación l, la única regla que puede derivar esta transición es *If2*. Por lo tanto, si su premisa es

$$\tilde{S}_i \;\vdash\; (l,\; \square\, B \mapsto P,\; l')$$

la sentencia G queda definida como:

$$G \;\doteq\; \square\, C_i \wedge B \mapsto P \;\;.$$

De la misma manera que en el caso anterior, analizaremos dos subcasos: $l' = \bullet$ y $l' \in \{e, c_1, \cdots, c_m\}$. En el primero, por hipótesis inductiva se cumple

$$\{I\} \,\square\, B \mapsto P \,\{q\}$$

con lo cual, por definición de G, se verifica la segunda proposición del lema. En el segundo subcaso ($l' \in \{e, c_1, \cdots, c_m\}$), por hipótesis inductiva se cumple

$$\{I\} \,\square\, B \mapsto P \,\{I\}$$

y de la misma manera, por definición de G, se verifica la cuarta proposición. Q.E.D. □

A partir del lema demostraremos el teorema 5.4.

Teorema
Sea un monitor con señalamiento automático que verifica el invariante I a partir de las condiciones de prueba

$$\begin{array}{l}\{\mathsf{true}\}\ \mathbf{enter}\ \{I\}\\ \{I\}\ \mathbf{await}.\mathsf{B}_i\ \{I \wedge \mathsf{B}_i\}\\ \{I\}\ \mathbf{return}\ \{\mathsf{false}\}\ .\end{array}$$

Sea $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ el sistema de transiciones generado a partir de la implementación del monitor. Entonces si $\Theta \subseteq I$, el invariante I es preservado en este sistema:

$$\mathsf{TS} \models \Box \varphi \quad \text{con} \quad \varphi.l \doteq I \quad (l \in \mathcal{L})\ .$$

Demostración
Sea S el cuerpo de un procedimiento del monitor con señalamiento automático. Como las primitivas **enter** y **return** aparecen al principio y al final respectivamente, el programa tiene la forma $S = \mathbf{return}; T; \mathbf{enter}$ con T el subprograma entre estas primitivas. Notemos que el programa T no contiene estas primitivas (solo aparecen al principio y al final de S) por lo cual cumple con las condiciones del lema anterior. Además, como S cumple las condiciones de prueba sobre **enter** y **return**, el programa T verifica:

$$\{I\}\ T\ \{I\}\ .$$

Sean $\tilde{S}$ y y $\tilde{T}$ las implementaciones, como monitor de señalamiento explícito, de S y T respectivamente (claramente se cumple $\tilde{S} = \mathbf{return}; \tilde{T}; \mathbf{enter}$). Notar que toda transición desde $\tilde{S}$ de la forma

$$\tilde{S} \vdash (l,\ G,\ l')$$

es derivada a partir de la aplicación de las reglas *Seq1*, *Seq2* o *Seq3* teniendo como premisas las obtenidas desde $\tilde{T}$ junto con las derivadas de las primitivas **enter** y **return** (por las reglas *Enter*, *ReturnEnter* y *ReturnWait*). Veremos a continuación que estas transiciones preservan el invariante I.
En el caso que ambas locaciones de la transición derivada desde $\tilde{T}$ sean distintas a la auxiliar, la única regla para derivar la transición en $\tilde{S}$ es *Seq3* aplicada dos veces (para las composiciones secuenciales con **enter** y **return**). Por lo tanto la transición de $\tilde{T}$ es preservada en $\tilde{S}$. Aplicando el lema anterior sobre T y $\tilde{T}$ se verifica

$$\{I\}\ G\ \{I\}\ .$$

por lo tanto la transición en $\tilde{S}$ preserva el invariante.

En el caso que alguna de las locaciones de la transición derivada desde $\tilde{T}$ sea igual a la auxiliar, para obtener aquella desde $\tilde{S}$ se deberán utilizar también las reglas *Seq1* y *Seq2* teniendo como premisas las transiciones derivadas desde $\tilde{T}$ más las producidas por las reglas *Enter*, *ReturnEnter* y *ReturnWait*. Las tres últimas generan transiciones que no modifican I ya que, como es invariante del monitor sin implementar (monitor automático), no contiene las variables de longitud de las colas. Por lo tanto, como por el lema anterior las transiciones

de $\tilde{T}$ preservan el invariante, en este caso las transiciones derivadas desde $\tilde{S}$ también lo harán.

Con este análisis por casos se puede deducir que toda transición derivada desde el monitor preserva el invariante. Por lo tanto en el sistema $\mathsf{TS} \doteq (\mathcal{L}, \mathcal{S}, \mathcal{T}, \Theta)$ se cumple

$$\Big\langle \forall\, l, G, l' \; : \mathcal{T}.l.G.l' : \; \{I\}\; G\; \{I\} \Big\rangle$$

o de forma equivalente

$$\{\varphi\}\; \mathcal{T}\; \{\varphi\} \quad \text{con} \quad \varphi.l \;\doteq\; I$$

Finalmente, por la hipótesis $\Theta \subseteq I$ del teorema y aplicando el teorema 2.14 (pág. 45), se demuestra

$$\mathsf{TS} \models \Box\, \varphi \ .$$

Q.E.D. □

Capítulo 6

Conclusiones y trabajos futuros

El trabajo de tesis doctoral presentado muestra la factibilidad de utilización de construcciones de alto nivel para la programación concurrente (regiones críticas condicionales y monitores con señalamiento automático), sin pérdidas importantes en la eficiencia. Este resultado tiene su relevancia en el contexto científico y tecnológico actual ya que provee de herramientas conceptuales útiles para pensar de manera simplificada los programas concurrentes. Hoy en día, resulta escasa la oferta de este tipo de herramientas brindada por los lenguajes de programación. Como evidencia de este hecho, la dificultad para resolver problemas utilizando monitores con señalamiento explícito (construcciones comúnmente implementadas en los lenguajes) se ve reflejada en la complejidad de la modelización desarrollada en nuestro trabajo: para utilizarlos, el programador debe tener en cuenta el comportamiento operacional no determinista del sistema, propio de esta clase de monitores, a partir de los distintos estados posibles de varias colas de procesos involucradas en su implementación. Con la implementación de monitores de señalamiento automático propuesta, este problema se elimina sin pagar costos excesivos en eficiencia.

Con respecto al marco metodológico utilizado, cabe señalar la formalización realizada de sistemas de transiciones y sus transformadores, utilizando lógica de alto orden dentro del enfoque denominado *shallow embedding* (sección 1.1). Esto permitió no solo definir estructuras que no poseen un carácter claramente recursivo (sistemas de transiciones), sino también, relacionar de manera homogénea aquellos objetos con la teoría de puntos fijos (capítulo 2).

Otro aspecto metodológico a destacar corresponde al uso conjunto de distintos probadores de teoremas para resolver el problema. Esta perspectiva ya fue investigada en [BNT05] donde se muestra que la combinación de herramientas semiautomáticas basadas en lógica de alto orden (Isabelle/HOL) junto con otras automáticas sobre un formalismo más restringido (SMT solvers), facilita el proceso manual de demostración[1]. En nuestro trabajo se evidencia que, utilizando las estrategias implementadas en las primeras herramientas, se puede lograr esta integración para resolver un problema de forma totalmente automática.

[1] En la versión actual de Isabelle se incluye la posibilidad de utilizar SMT sovers a la manera de oráculos.

El enfoque utilizado para resolver el problema está inscripto dentro de la línea conceptual de la tradición inaugurada por Dijkstra [Dij76], donde se utiliza la lógica para construir programas y no solo para verificarlos a posteriori. Nuestro trabajo pone de manifiesto la utilidad de los métodos formales, no solo para verificar programas ya construidos, si no para generarlos a partir de construcciones más abstractas. De esta manera, las regiones críticas condicionales y los monitores de señalamiento automático pueden pensarse como especificaciones de problemas de concurrencia, desde las cuales derivamos implementaciones eficientes utilizando la lógica.

De todos modos, el uso de las técnicas desarrolladas en nuestro trabajo estuvo orientado a la optimización de programas. En su aplicación se pudo observar que los invariantes generados (los cuales validan las mejoras) son más fuertes que las condiciones de prueba iniciales (invariantes candidatos). Este fenómeno abre la posibilidad de utilizarlos, no solo para mejorar la eficiencia, sino también para comprender y analizar los programas concurrentes. Dentro de esta perspectiva es posible pensar en el desarrollo de una informática experimental, donde los métodos formales, apoyados por demostradores de teoremas y otras herramientas, sirvan para construir programas a partir de su análisis profundo. Creemos que esto será útil al momento de reducir los errores conceptuales en los programas concurrentes, lo cual en este aspecto, resulta más efectivo que la verificación a posteriori o la validación de caso de test sobre los mismos. Claramente el enfoque propuesto necesita de una mayor intervención por parte de los programadores al momento de desarrollar los programas. De todas maneras, debido a la complejidad intrínseca que muestra la programación concurrente, este esfuerzo es necesario si se desea obtener niveles de corrección aceptables. El mismo puede ser menguado con el mejoramiento de entornos de programación que vinculen de manera simple los probadores de teoremas a un proceso más mecanizado de desarrollo de programas.

En los capítulos anteriores se señalaron algunas líneas de investigación futuras específicas de los temas tratados en esas oportunidades. En los capítulos 4 y 5 se propone incluir a los métodos propuestos, técnicas de interpretación abstracta sobre distintos dominios, analizando también la posibilidad del desarrollo de operadores de narrowing sobre el cálculo del máximo punto fijo en estos dominios. En el capítulo 5 además se menciona como posible trabajo futuro la implementación de monitores con señalamiento automático mediante la adaptación de la técnica de Semáforos Binarios Divididos. Como se mostró en aquella oportunidad, los sistemas de transiciones que modelan problemas implementados con esta técnica, son de orden de magnitud inferior en tamaño a los implementados con monitores. A partir de esta evidencia, creemos que este nuevo problema puede ser resuelto en tiempos de computo menores, permitiendo así escalar la propuesta a programas de mayor tamaño.

Como ya mencionamos en estas conclusiones y en la sección 4.9, otra importante línea de investigación es el estudio y desarrollo de entornos integrados de programación que incluyan la generación automática de invariantes como guía para el desarrollo de programas.

Índice alfabético

$\langle _ \rangle^{\sharp}$, 96
$\cap$, 5
$[_]^{\sharp}$, 95
$\wedge$, 2
$\mathcal{B}_{\mathcal{T},P}$, 54
$=$, 5
$\equiv$, 2, 5
$\mathcal{F}_{\mathcal{T},\Theta}$, 48
$\Rightarrow$, 2, 5
$\sqsubseteq^{\sharp}$, 68
$\neg$, 2, 5
$\cup$, 5
$\vee$, 2
$\mathcal{P}^{\sharp}$, 68
$\subseteq$, 5
$\bot$, 5, 8
$\Cup$, 91
$[\![\ ]\!]$, 2
$\dot{\sigma}$, 95
$[_]$, 9
Σ, 4
$\langle _ \rangle$, 9
$\mathcal{L}$, 17
$\bullet$, 164
γ, 68
$\mu.\mathcal{F}_{\mathcal{T},\Theta}$, 48
$\nu.\mathcal{B}_{\mathcal{T},P}$, 53
$\top$, 5
$\triangleq$, 7
$\vdash$, 131
∇, 71
$\sqcap$-semireticulado, 39
$\sqcup$-semireticulado, 39
$\sqcap$-continua, 42
$\sqcup$-continua, 41

abort, 10
abstracción, 69
abstracción de, 69
actualización funcional, 9
actualización funcional abstracta, 96
adyacencia, 85
and-continuo, 14
anidado, 35
asignación guardada, 10, 22
asignación lineal, 94
asignación múltiple, 10

base, 85
base factible, 85
bases adyacentes, 85
bottom, 8
bounded quantification, 3
busy wait, 104, 152

cambio de variable, 35
cara de un poliedro, 85
Card, 2
cilindro, 78
co-cpo, 40
cola de procesos, 154
combinación lineal, 77
combinación lineal convexa, 77
combinación lineal positiva, 77
composición secuencial, 9
cond, 154
configuración, 18, 28
conical hull, 77
conjunto codirigido, 14
conjunto de índices, 5
conjunto de configuraciones, 28
conjunto de configuraciones iniciales, 18
conjunto de estados alcanzables, 8
conjunto de locaciones, 17
conjunto dirigido, 14
contadores SBD, 108
continuidad, 14

Convex hull, 77
convexo, 75
cpo, 40
cumple, 6
CVC3, 104

deadlock, 108
deep embedding, 1
definiciones locales, 2
describe, 6
desigualdad lineal, 75
dimensión de la variedad lineal, 78
dimensión de un poliedro convexo, 78
Disk schedulling, 170, 201
Dom, 2
dominio semántico abstracto, 68
dominio semántico concreto, 67

ejecución, 29
ejecución finita, 28
ejecución infinita, 29
envoltura cónica, 77
envoltura convexa, 77, 89
envoltura convexa de un marco, 79
equivalente, 7
espacio afín, 78
espacio de estados, 4
espera condicional, 104
estado, 4
estado abierto, 154
estado cerrado, 153
estado/locación, 18
estricta, 41
estricto, 13
expresión cuantificadas, 3
Expresiones booleanas lineales, 93
expresiones existenciales, 7
expresiones universales, 7

falso, 6
forma canónica, 85
forma estándar, 84
forma normal disyuntiva, 94
Fun, 9
función de concretización, 68
función semántica abstracta, 69
función semántica concreta, 68

gráfico de transiciones, 20

havoc, 171
hiperplano, 78

ínfimo de poliedros, 89
interpretación abstracta, 151
interrumpible, 13, 41, 54
intersección de poliedros, 89
invariante, 44, 47
invariante candidato, 47
invariante de problema RCC, 108
invariante del monitor, 160
invariante inductivo, 44
invariante inductivo candidato, 44, 53

juntividad, 13

k-cara, 85
k-inducción, 61
k-invariante, 62
Knaster, 42

línea, 78
locación auxiliar, 164
locación de llegada, 17
locación de salida, 17
locaciones iniciales, 18

más débil, 6
más fuerte, 6
método de base artificial, 85
método simplex, 85
métodos de no-pivoting, 84
métodos de pivoting, 84
marco, 79
monótona, 41
monótono, 13
monitores con señalamiento automático, 151
monitores con señalamiento explícito, 151
monotonía, 13

NewPolka, 102
no interrumpible, 13, 41
x Non-Blocking, 155
No Priority Restricted Automatic Signal, 158
no satisface, 6
no satisfactible, 6
nonaborting, 13
NPNB, 155
NPRAS, 158, 160

operador de elección, 2, 9
operador de widening, 71
or-continuo, 14

partición de rango, 34
PB, 155
pila, 170
poliedro acotado, 78
poliedro convexo cerrado, 76
poliedros convexos, 151
poliedros convexos cerrados, 74
Poly, 76
positivamente conjuntiva, 41
positivamente conjuntivo, 13, 41
positivamente disyuntiva, 41
positivamente disyuntivo, 41
Pred, 5
predicado, 5
predicado de un sistema de transiciones, 18
$\mathsf{Pred}_{\mathcal{L}}$, 18
Priority Blocking, 155
problema del convex hull, 83
proceso activo, 154
Productor/Consumidor, 110
propagación hacia adelante, 48
propagación hacia atrás, 54
proyección, 80

rango de especificación, 3
rango unitario, 34
rango vacío, 34
rayo, 77
rayo extremo, 77
recta, 78
regiones críticas condicionales, 105
regla de intercambio, 36
regla del dominó, 182
regla del término, 34
regla del término constante, 35
relación de transición, 28
rendezvous, 197

satisface, 6
satisfactible, 6
saturación, 83
SBD, 103, 105
SBD conjunto, 107
SBD técnica, 106
scheduler de monitor, 154
sección, 79, 119
semáforo neutral, 107
semáforos binarios divididos, 105, 106
semáforos generales, 107
semántica de interleaving, 118
semántica de trazas, 68
semántica estándar, 67
semántica operacional, 68
semiespacio cerrado, 75
Sent, 8
sentencia determinista, 9
sentencia lineal, 94
sentencia no termina, 8
sentencia no termina desde, 8
sentencia total, 9
sentencias, 8
shallow embedding, 1, 5, 7, 10, 217
signal., 154
Signal and Urgent Wait, 155, 159
Signal and Wait, 154
signal as hint, 156, 161, 162, 166, 174
sincronización de fase, 190
sistema de restricciones lineales, 76
sistema de transiciones, 18
sistemas de transiciones, 1
skip, 10
SMT solver, 103
Standard ML of New Jersey, 102, 137, 184
strongest postcondition, 11, 23
subespacio lineal generado, 77
suposición, 9, 10
suposición abstracta, 95
suposición lineal, 94
supremo de poliedros, 89

término de cuantificación, 3
Términos aritméticos lineales, 93
Términos booleanos lineales, 93
Tarski, 42
terminating, 13
$\mathsf{Tran}_{\mathcal{L},\mathcal{S}}$, 17
transición entrante, 18
transición saliente, 18
transiciones, 17

universalmente conjuntivo, 14, 41
universalmente disyuntivo, 14, 41

válida, 6

vértice, 77
vértices adyacentes, 85
variable de cuantificación, 3
variables aumentadas, 84
variables de condición, 154
variables iniciales, 84
variedad lineal, 78
variedad lineal generada, 78
variedad lineal maximal, 78
variedad ortogonal, 78
verdadero, 6

wait., 154
Wait and Notify, 154, 159
weakest liberal precondition, 12, 24
weakest precondition, 12
widening de poliedros, 100
widening estándar, 100

Bibliografía

[ACL08] ACL2 home page. http://www.cs.utexas.edu/users/moore/acl2, 2008. 131

[And89] Gregory R. Andrews. A method for solving synchronization problems. *Sci. Comput. Program.*, 13(1):1–21, 1989. 104

[And91] Gregory R. Andrews. *Concurrent programming: principles and practice.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991. 107, 157, 162, 170, 193, 194, 201

[And99] G. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, Reading, Massachusetts, USA, 1999. 107

[AO97] Krzysztof R. Apt and Ernst-Rüdiger Olderog. *Verification of sequential and concurrent programs (3rd ed.).* Springer-Verlag New York, Inc., 1997. 211, 213

[BAW98] Ralph-Johan J. Back, Abo Akademi, and J. Von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998. 1, 2, 3, 5, 8, 10, 13, 16, 21, 42, 43, 183, 186

[BB07a] Damián Barsotti and Javier O. Blanco. Automatic refinement of split binary semaphore. In Cliff B. Jones, Zhiming Liu, and Jim Woodcock, editors, *Theoretical Aspects of Computing - ICTAC 2007*, number 4711 in Lecture Notes in Computer Science, pages 64–78. 4th International Colloquium Macao,China, Springer, September 2007. http://www.cs.famaf.unc.edu.ar/~damian/publications/ictac2007/sbd.pdf. 104, 107

[BB07b] Damián Barsotti and Javier O. Blanco. Generación de invariantes para implementar eficientemente regiones críticas condicionales. In *Proceedings of CACIC 2007*, pages 1690–1701. XIII Congreso argentino de ciencias de la computación, October 2007. http://www.cs.famaf.unc.edu.ar/~damian/publications/cacic2007/sbd.pdf. 104

[BBM97] Nikolaj Bjorner, Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theor. Comput. Sci.*, 173(1):49–87, 1997. 47, 64, 102

[BFC95] Peter A. Buhr, Michel Fortier, and Michael H. Coffin. Monitor classification. *ACM Comput. Surv.*, 27(1):63–107, 1995. 154, 155, 157, 158, 159, 161, 171

[BH05] Peter A. Buhr and Ashif S. Harji. Implicit-signal monitors. *ACM Trans. Program. Lang. Syst.*, 27(6):1270–1343, 2005. 155, 159, 161, 162, 201

[BHRZ03] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. Quaderno 312, Dipartimento di Matematica, Università di Parma, Italy, 2003. `http://www.cs.unipr.it/Publications/Abstracts/Q312`. 100

[Bid07] Natalia Beatriz Bidart. Generación automática de invariantes lineales. Bachelor's Degree thesis, Facultad de Matemática Astronomía y Física, Universidad Nacional de Córdoba, 2007. Supervisor Damián Barsotti. 102

[BJS90] Mokhtar S. Bazaraa, John J. Jarvis, and Hanif D. Sherali. *Linear programming and network flows (2nd ed.)*. John Wiley & Sons, Inc., New York, NY, USA, 1990. 84

[BL05] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '05, pages 82–87, New York, NY, USA, 2005. ACM. 171, 201

[Blu01] Matthias Blume. No-longer-foreign: Teaching an ml compiler to speak c "natively". *Electr. Notes Theor. Comput. Sci.*, 59(1), 2001. 186

[BNT05] Damián Barsotti, Leonor Prensa Nieto, and Alwen Tiu. Verication of clock synchronization algorithms: Experiments on a combination of deductive tools. In *Proceedings of AVOCS 2005*, volume 145 of *ENTCS*, pages 63–68. Elsevier Science B. V., 2005. `http://www.cs.famaf.unc.edu.ar/~damian/publications/clock.pdf`. 217

[BSB08] Javier Blanco, Silvina Smith, and Damián Barsotti. *Cálculo de programas*. Universidad Nacional de Córdoba, 2008. `http://www.cs.famaf.unc.edu.ar/~damian/calcprog`. 1, 4, 31, 32, 34

[BT07] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19th *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany. 104

[BW03] Javier Blanco and Nicolás Wolovick. Strengthen, widen, get semaphores. In *WAIT2003*, Buenos Aires, Argentina, september 2003. 32 JAIIO. `http://www.cs.famaf.unc.edu.ar/~nicolasw/Publicaciones/strengthenwiden.pdf`. 190, 191

[BW10] Damián Barsotti and Nicolas Wolovick. Automatic probabilistic program verification through random variable abstraction. In *Eighth Workshop on Quantitative Aspects of Programming Languages (QAPL 2010)*, 2010. 149, 201

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977. 67

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2:511–547, 1992. 67, 68, 69

[CGJ+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement, 2000. 149, 201

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY. 84, 100

[Che64] N. V. Chernikova. Algorithm for finding a general formula for the non-negative solutions of system of linear equations. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 4(4):151–158, 1964. 88

[Chv83] Vašek Chvátal. *Linear Programming.* A Series of Books in the Mathematical Sciences. Freeman, 1983. 75, 84, 85

[Cla79] Edmund Melson Clarke, Jr. Synthesis of resource invariants for concurrent programs. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 211–221. ACM Press, 1979. 117, 119

[Cre05] Mache Creeger. Multicore CPUs for the masses. *ACM Queue*, 3(7):64–ff, september 2005. 106, 148

[CVC08] CVC3 home page. `http://www.cs.nyu.edu/acsys/cvc3`, 2008. 104, 186

[DD02] Satyaki Das and David L. Dill. Counter-example based predicate discovery in predicate abstraction. In *In Formal Methods in Computer-Aided Design*. Springer, 2002. 149, 201

[Dij68] Edsger W. Dijkstra. Cooperating sequential processes. `http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF`, 1968. 144

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice Hall Series in Automatic Computation. Prentice Hall, 1976. 218

[Dij79] Edsger W. Dijkstra. A tutorial on the split binary semaphore. Circulated privately `http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD703.PDF`, March 1979. 103, 105, 107, 117, 119, 144, 201

[Dij80a] Edsger W. Dijkstra. A notational alternative for quantification. Circulated privately http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD737.PDF, April 1980. 4

[Dij80b] Edsger W. Dijkstra. The superfluity of the general semaphore. Circulated privately http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD734.PDF, April 1980. 107

[Dij00] Edsger W. Dijkstra. The notational conventions i adopted, and why. Circulated privately http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1300.PDF, July 2000. 4

[Dijnd] Edsger W. Dijkstra. Over seinpalen. Circulated privately http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD74.PDF, n.d. 106

[DP90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990. 41

[DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate calculus and program semantics*. Springer-Verlag New York, Inc., 1990. 1, 10, 12, 13, 16, 26, 39, 171

[EJL+02] Johan Eker, Jorn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity—the ptolemy approach. In *Proceedings of the IEEE*, volume Special Issue on Modeling and Design of Embedded Software, October 2002. 106

[Fou41] Charles Fourier. *Oeuvres completes de Ch. Fourier*. Paris : Aux Bureaux de la Phalange., 2nd ed. edition, 1841. 81

[FQ88] F. Fernandes and P. Quinton. Extension of chernikova's alghorithm for solving general mixed linear programming problems. Technical Report 437, IRISA, Rennes, France, 1988. 88

[Gib94] W. Wayt Gibbs. Software's chronic crisis. *Scientific American*, page 86, September 1994. 106, 148

[Hal06] Nicolas Halbwachs. On the design of widening operators. VMCAI'06: 7th International Conference on Verification, Model Checking, and Abstract Interpretation, 2006. Tutorial. 100

[Han72] Per Brinch Hansen. Structured multiprogramming. *Commun. ACM*, 15(7):574–578, 1972. 105, 152

[Han73a] Per Brinch Hansen. Concurrent programming concepts. *ACM Comput. Surv.*, 5:223–245, December 1973. 152

[Han73b] Per Brinch Hansen. *Operating system principles*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1973. 161

[Han81] Per Brinch Hansen. The design of edison. *Softw., Pract. Exper.*, 11(4):363–396, 1981. 160, 161

[Hoa72] C.A.R. Hoare. Towards a theory of parallel programming. In C.A.R. Hoare and R.H. Perrott, editors, *Operating System Techniques*, pages 61–71. Academic Press, 1972. 105

[Hoa74] C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974. 105, 117, 152, 157, 161, 201

[Hoo86] Rob Hoogerwoord. Split binary semaphores. Manuscript RH86, 1986. 107

[Hoo90] Rob Hoogerwoord. Two applications of the split binary semaphore. Course notes RH221, 1990. 107, 111

[HPR97] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2), August 1997. 89, 186

[Jea05] Bertrand Jeannet. *The POLKA Convex Polyhedra library*, 2.1.0 edition, June 2005. 102

[Kuh56] H. W. Kuhn. Solvability and consistency for linear equations and inequalities. *American Mathematical Monthly*, 63:217–232, 1956. 81

[Lan66] E. Lanery. Recherche d'un système générateur minimal d'un polyhedre convexe, 1966. Thèse de 3ème cycle. 83, 84, 85, 87

[Lee06] Edward A. Lee. The problem with threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, January 10 2006. 106, 148

[LR80] Butler W. Lampson and David D. Redell. Experience with processes and monitors in mesa. *Commun. ACM*, 23:105–117, February 1980. 156

[MM04] Annabelle McIver and Carroll Morgan. *Abstraction, Refinement And Proof For Probabilistic Systems (Monographs in Computer Science)*. SpringerVerlag, 2004. 8

[Mon00] D. Monniaux. Abstract interpretation of probabilistic semantics. In Jens Palsberg, editor, *SAS*, volume 1824 of *Lecture Notes in Computer Science*, pages 322–339. Springer, 2000. 69

[Mot36] T. S. Motzkin. *Beiträge zur Theorie der Linearen Ungleichungen.* PhD thesis, Berlin, 1936. 81

[MP91] Zohar Manna and Amir Pnueli. On the faithfulness of formal models. In *Mathematical Foundations of Computer Science*, pages 28–42, 1991. 118, 120

[MRS93] M.W. Mislove, A.W. Roscoe, and S.A. Schneider. Fixed points without completeness. *Theoretical Computer Science*, 138:138–2, 1993. 41, 43

[MRTT53] T. S. Motzkin, H. Raiffa, G. L. Thompson, and R. M. Thrall. The double description method. In H. W. Kuhn and A. W. Tucker, editors, *Contributions to the Theory of Games – Volume II*, number 28 in Annals of Mathematics Studies, pages 51–73. Princeton University Press, Princeton, New Jersey, 1953. 88

[MvdS89] A.J. Martin and J.L.A. van de Snepscheut. Design of synchronization algorithms. *Constructive Methods in Computing Science*, pages 445–478, 1989. 107

[NPN99] Tobias Nipkow and Leonor Prensa-Nieto. Owicki/Gries in Isabelle/HOL. In J.-P. Finance, editor, *Fundamental Approaches to Software Engineering (FASE'99)*, volume 1577 of *LNCS*, pages 188–203. Springer, 1999. 1

[OG76] Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6:319–340, 1976. 190

[Pau94] Lawrence C. Paulson. *Isabelle, A Generic Theorem Prover*, volume 828 of *LNCS*. Springer-Verlag, 1994. 104, 186

[Sch76] Hans Albrecht Schmid. On the efficient implementation of conditional critical regions and the construction of monitors. *Acta Informatica*, 6(3):227–249, August 1976. 117, 118, 143, 161

[Sch97] Fred B. Schneider. *On Concurrent Programming*. Graduate texts in computer science. Springer-Verlag New York, Inc., 1997. 107

[Sta06] Standard ML of New Jersey home page. `http://www.smlnj.org/`, 2006. 102, 138, 184

[Tea02] The Polylib Team. *Polylib User's Manual*, September 26 2002. 89

[TRSS01] A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar. A technique for invariant generation. In Tiziana Margaria and Wang Yi, editors, *TACAS 2001 - Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031, pages 113–127, Genova, Italy, April 2001. Springer-Verlag. 64

[Ver92] H. Le Verge. A note on chernikova's algorithm. Publication interne 635, IRISA, Campus de Beaulieu, Rennes, France, July 27 1992. 88

[Wil93] Doran K. Wilde. A library for doing polyhedral operations. Publication interne 785, IRISA, Campus de Beaulieu, Rennes, France, December 1993. 89

[yic08] Yices home page. `http://yices.csl.sri.com`, 2008. 104